The LLNL High Accuracy Volume Renderer for Unstructured Data: Capabilities, Current Limits, and Potential for ASCI/VIEWS Deployment

Peter L. Williams and Nelson L. Max

Lawrence Livermore National Laboratory
Livermore, CA 94551

Abstract

This report describes a volume rendering system for unstructured data, especially finite element data, that creates images with very high accuracy. The system will currently handle meshes whose cells are either linear or quadratic tetrahedra, or meshes with mixed cell types: tetrahedra, bricks, prisms, and pyramids. The cells may have nonplanar facets. Whenever possible, exact mathematical solutions for the radiance integrals and for interpolation are used. Accurate semitransparent shaded isosurfaces may be embedded in the volume rendering. For very small cells, subpixel accumulation by splatting is used to avoid sampling error. A new exact and efficient visibility ordering algorithm is described. The most accurate images are generated in software, however, more efficient algorithms utilizing graphics hardware may also be selected. The report describes the parallelization of the system for a distributed-shared memory multiprocessor machine, and concludes by discussing the system's limits, desirable future work, and ways to extend the system so as to be compatible with projected ASCI/VIEWS architectures.

1 Introduction

This report describes the LLNL high accuracy (HIAC) volume rendering system for unstructured data, especially finite element data, that, for a given mathematical optical model [27], creates images in software with very high accuracy. Whenever possible, exact mathematical solutions for the differential equations involved and for interpolation are used. The data is used at its original vertex positions, and is not resampled. The rendering is based on cell projection which allows the use of subpixel accumulation by splatting to avoid sampling

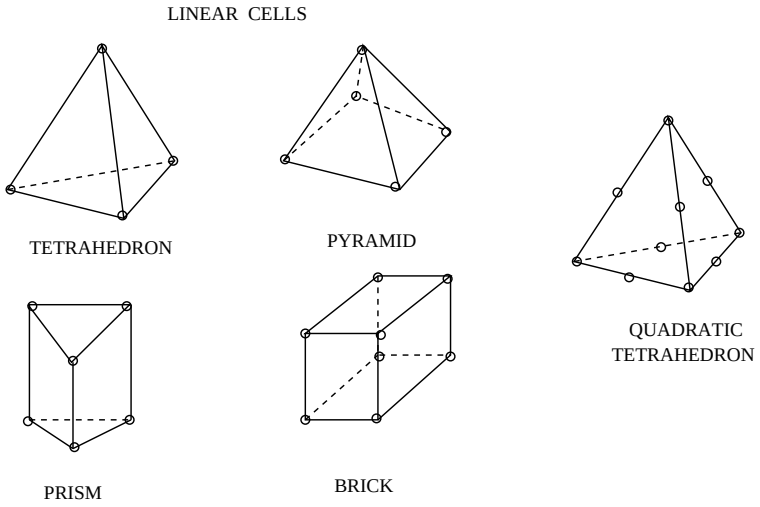

Figure 1: Examples of different types of cells used in the finite element method. The HIAC system will render data defined on meshes with any of these cell types.

error. Accurate semitransparent shaded isosurfaces may be embedded in the volume rendering. This volume rendering system, a much evolved version of the system reported in 1990 by Max, Hanrahan and Crawfis in [26], is based on the absorption plus emission optical model [27, 42, 60] and utilizes the cell projection method to accumulate the image. A ray integration is performed individually for every pixel onto which a cell projects. The system will correctly render images in both parallel and perspective projection provided the transfer functions for color and opacity are piecewise linear.

HIAC is intended primarily for data sets from the finite element method, but will render any unstructured data set whose cells are tetrahedra, bricks, prisms, or pyramids, or any combination thereof; see Fig. 1. The meshes may be nonconvex or even disconnected; the faces of adjacent cells may meet on only part of their common adjacent face, i.e. sliding interfaces are permitted. However, the cells are expected to be nonintersecting, and the visibility ordering graph should not contain cycles (the mesh is *acyclic*).

The system will render with the highest accuracy data sets with linear or quadratic tetrahedra. For linear tetrahedra, (tetrahedra where the scalar field varies linearly within the cell), the system uses the exact solution to the radiance integral described by Williams and Max in [60], using the Dawson integral [39]. Quadratic tetrahedra, (tetrahedra where the scalar field varies quadratically along the edges of the cells, and, in fact, along any ray through the cell), are also rendered accurately as described in Section 5.2.1.

The system will also render meshes with mixed cell types (*zoo* elements): tetrahedra, bricks, prisms, and pyramids, however, nontetrahedral elements are currently not rendered with the same high accuracy as tetrahedra. Instead, nontetrahedral cells are subdivided before rendering as described in Section 5.3. This report describes in detail how HIAC could be extended to render such nontetrahedral elements with high precision. The cells may have nonplanar faces, which occur in deformed and curvilinear meshes, and in meshes from the finite element method where cells can be deformed by a parametric mapping function.

Provided the cells are not highly distorted (see Section 11.2), HIAC will render them.

In addition to the high accuracy modes described above, which utilize software rendering, the HIAC system allows the user to select one of several different lower accuracy volume rendering approximations which are significantly faster. Some of these approximations take advantage of graphics hardware. These approximations are described in more detail in Sections 5.3 and 6.

Typically unstructured meshes have a complex geometric configuration and the mathematics of the absorption-emission integral are quite complex. Therefore most existing volume rendering systems for unstructured data [14, 15, 16, 17, 20, 22, 23, 25, 26, 38, 46, 48, 50, 55, 56, 57, 59, 64, 67] introduce various simplifying assumptions and approximations into the algorithm, or resample the data onto a rectilinear mesh, in order to cope with these complexities in an efficient manner.

Another aspect of unstructured meshes is that typically they are adaptively refined, so that in areas where the field is changing rapidly, the cells are smaller than in other areas of the mesh. It is not uncommon for such cells to be several orders of magnitude smaller than the largest cells. The behavior of the field on these smallest cells is often of great interest to the simulation scientist. However, most volume rendering systems for unstructured data are liable to miss these smaller cells due to sampling error. Some systems, such as the system created by Mark Duchaineau referred to in Section 2, use hierarchical resampling onto a rectilinear mesh to conserve memory and still retain the data in the highly refined areas of the mesh, however, the original mesh structure is lost and therefore so is the ability to do interpolation on the original cells.

Our original goal was to design a volume rendering system to create benchmark images for use as a standard of comparison. The benchmarks can be used to compare results from other volume rendering systems for unstructured data that use approximations, simplifying assumptions, and/or resampling, and they can serve as a validation suite for verifying the correctness of new algorithms and implementations. Subsequently, we amended our goal to generate volume rendered images of zoo mesh data and cells with nonplanar facets, with some sacrifice in accuracy, and to take advantage of OpenGL hardware rendering, for greater speed.

HIAC uses a new, exact and efficient, visibility sorting algorithm [5], called the *Scanning Exact Meshed Polyhedra Visibility Ordering* (SXMPVO) algorithm, developed specifically for this system, which handles nonconvex cells, disconnected meshes and cells which have a sliding interface. It does however require the mesh to be acyclic. This algorithm is described in detail in Section 8. For interactive viewing, the faster *Meshed Polyhedra Visibility Ordering — Nonconvex* (MPVONC) algorithm [58] may also be used, but it is not guaranteed to be correct.

In addition to giving an in-depth description of the above system and its capabilities, this report also describes how the system has been parallelized for a distributed-shared memory multiprocessor machine with graphics hardware. It further discusses the boundaries and limits of the HIAC volume rendering system, desirable future work, and how the system

might be modified for use in projected ASCI/VIEWS architectures.

The next section discusses related previous work. Section 3 discusses the geometry of the cells used in the finite element method, the related interpolation equations, and relevant terminology. Section 4 gives a broad overview of the system, and then in Section 5 we cover the details of the volume rendering engine. Section 6 discusses the hardware-based rendering algorithms. Section 7 discusses an early termination scheme alternative. In Section 8 we present the details of the rendering system and of the visibility ordering algorithm. Section 9 discusses the parallelization and load balancing algorithms used by the HIAC system. Section 10 presents timing results and example images. Section 11 covers the limits of the HIAC volume rendering system and desirable future work. Section 12 addresses how the system might be modified so as to be useful in the context of projected future architectures for ASCI/VIEWS.

2 Related Work

A method for approximating the volume rendering integral with bounded error is described by Novins and Arvo in [36]. By bounding the magnitude of the derivatives of the integrand, they are able to obtain remainder terms that provide bounds on the approximation error. They apply this to the trapezoid rule, Simpson's rule, and a power series method. The first two methods are more suited to low to medium accuracy approximations. The power series method on the other hand is preferable for very high precision results, but is slow.

The techniques developed by Novins and Arvo are very valuable for bounding the error in the evaluation of the integral. However, there are other sources of error in the volume rendering process, e.g. sampling error which may miss small but highly important cells in the accumulation process, that may be even more significant than integration error. The HIAC system addresses some of these other sources of error. For example, it uses subpixel splatting and a high accuracy visibility ordering algorithm.

For high accuracy integration, the HIAC system uses a closed form solution to the integral when possible, otherwise high accuracy Gaussian numerical integration is used. This approach appears to be more efficient than the power series method, since the power series error bounds are loose. (Novins did not provide timing data for comparative purposes.) The error bounds are not easy to calculate for Gaussian quadrature, but it is known to be very accurate for sufficiently smooth functions and it is the quadrature method generally used in the finite element method. When a guaranteed error bound is required for integration on higher order elements, the Novins and Arvo power series approximation may be valuable. It is an open question whether Gaussian quadrature or the Power Series method described by Novins is preferable for high accuracy integration.

Silva and Mitchell [48] describe a very efficient and interesting sweep plane volume rendering method that accurately traverses all types of tetrahedral meshes with nonintersecting cells, even those with cyclically overlapping cells. They claim it can be extended in a straightforward way to more complex convex cells. The real value of the sweep plane algorithm is that

it provides a very efficient and accurate depth ordering of the cells of an unstructured mesh along any given ray to the eye; it does not try to give a global visibility ordering of the cells. The mathematics of the volume rendering integral is not addressed in their report, nor is sampling error. The integration methods described here could be utilized in the sweep plane algorithm.

Gallagher and Nagtegaal [13] describe methods for rendering 3D contour surfaces of finite element data as well as methods for smooth shading these surfaces. They render the contour surfaces, which may be curved within a cell, as a polygonal approximation to a parametric bicubic surface fit to each contour in a cell, whereas, for quadratic elements, we render these same surfaces on a pixel by pixel basis to reproduce the exact implicit curved surface and use Phong shading calculated at each pixel. Cline et al. [3] also reproduce this curved surface by recursive subdivision of the volume cells containing the contour surface.

Mark Duchaineau at LLNL has developed an unpublished method for hierarchical adaptive resampling of unstructured meshes to a rectilinear grid which uses wavelet compression. This has proven very useful for rapidly visualizing terascale data sets, including volume rendering such data. The only drawback to this method is that the original mesh is lost and if the domain scientist wishes information related to the original mesh then this is not possible. It will be interesting to compare volume rendered images created by this resampling system with images of the same data set created by HIAC.

Relevant previous work in the area of visibility ordering of unstructured meshes is discussed in Section 8. Previous work that contains ideas that might be useful to incorporate into HIAC is covered in Section 11.6.7.

3 Cell Geometry and Interpolation Functions

The cells used for 3D modeling in the finite element method (FEM) have many different shapes, but only a few are in widespread use [24]. We focus our attention on the more commonly used 3D cells (also called *elements*): the tetrahedron, brick and prism. See Fig. 1.

In addition to the vertices (also referred to as *nodes*) used to define the endpoints of a cell's edges, which we will call the *conventional vertices* or *nodes*, a cell may have additional vertices, which we will call *interior nodes*, see for example the quadratic tetrahedron in Fig. 1. The interior nodes, along with the conventional nodes, may be used: (a) to define a nonlinear field inside the cell by the use of what we will refer to as an *interpolation function*; and/or (b) to define curvilinear facets by the use of a parametric mapping function. When the interior nodes are used only to define the geometry, the element is called a *superparametric* element, when used only to define the scalar field, a *subparametric* element, or if used for both an *isoparametric* element.

In this report, we will not deal with elements whose geometry is defined by a parametric mapping since those elements may have facets that are highly curved, and the parametric mapping must be inverted before the scalar function can be evaluated. We will limit our

consideration to subparametric cells. For those cells, the scalar field value is specified at all vertices, conventional and interior; but the geometry of the cell is determined from its conventional vertices.

The number of terms in a cell's interpolation function is equal to the number of nodes that the cell has. So a tetrahedron with four nodes will have an interpolation function with four terms. In most applications, the interpolation function is a polynomial whose terms are elements of the three dimensional power series. Those terms through third degree are:

$$\begin{gathered} 1 \\ x, y, z \\ x^2, y^2, z^2, xy, xz, yz \\ x^3, y^3, z^3, x^2y, x^2z, xy^2, xz^2, y^2z, yz^2, xyz. \end{gathered} \tag{1}$$

The interpolation function for a 4-node tetrahedron is:

$$f(x, y, z) = c_1 + c_2x + c_3y + c_4z.$$

The scalar field varies linearly along any ray through a 4-node tetrahedron, hence it is called a linear tetrahedron.

A brick with eight nodes has the eight term interpolation function:

$$f(x, y, z) = c_1 + c_2x + c_3y + c_4z + c_5xy + c_6xz + c_7yz + c_8xyz. \tag{2}$$

The particular terms of the 3D power series that are chosen for a given interpolation function are dictated by the need of the FEM for certain desirable properties such as symmetry, nonsingularities, etc. Here, the scalar field varies linearly along the edges of the brick and so it is sometimes called a linear brick. However, the field inside the brick varies trilinearly so it is also called a trilinear brick. Others refer to it as an 8-node brick, or a hexahedron. Often it is the case that nontriangular facets are nonplanar.

Although the 4-node tetrahedron and the 8-node brick are both referred to as linear elements, the higher order terms in the interpolation equation for the linear brick give it extra degrees of freedom that allow it to solve some problems much more accurately than could be done with tetrahedra alone. From the perspective of visualization, it should be noted that a contour surface inside an 8-node brick is curved and not planar as it is inside a 4-node tetrahedron, and also that the variation of the scalar function along a general ray is cubic rather than linear.

The interpolation function for the 6-node pentahedron or prism is:

$$f(x, y, z) = c_1 + c_2x + c_3y + c_4z + c_5xz + c_6yz \tag{3}$$

The prism is mainly used as a transition element to "glue" together tetrahedra and bricks in meshes that use a combination of different element types.

The pyramid element, see Figure 1, occasionally used in the FEM and appearing in curvilinear meshes, has an interpolation function that is generated from the interpolation function

of the brick element by assigning the same nodal coordinates and scalar values to more than one node.

The tetrahedron, brick and prism are the basic cells. They are often referred to as linear cells since the field varies linearly along the edges of the cells. By adding interior nodes to the basic cells we get cells with higher order interpolation functions. We refer to this class of cells as *higher order cells*. There are three important higher order cells.

The first is the 10-node tetrahedron, also referred to as a quadratic tetrahedron, whose interpolation function is:

$$f(x,y,z) = c_1 + c_2x + c_3y + c_4z + c_5x^2 + c_6y^2 + c_7z^2 + c_8xy + c_9xz + c_{10}yz. \tag{4}$$

This function is complete through the quadratic terms of the 3D power series in (1), therefore the field varies quadratically along any ray through the volume. In the FEM, the six interior nodes may be specified in different configurations, however the most common configuration is for the interior nodes to be located on the edges of the cell, usually at the midpoints. Elements where all of the nodes lie on the boundary of the element are called *serendipity* elements. Serendipity elements are the most common 3D elements.

The next higher order element is the cubic tetrahedron which has 20 nodes. This cell has two interior nodes per edge, usually at $\frac{1}{3}$ and $\frac{2}{3}$ of the edge, as well as a node at the center of gravity of each facet of the cell. Its interpolation function is:

$$\begin{aligned} f(x,y,z) \;=\; & c_1 + c_2x + c_3y + c_4z + c_5x^2 + c_6y^2 + c_7z^2 + c_8xy + c_9xz + c_{10}yz + \\ & c_{11}x^3 + c_{12}y^3 + c_{13}z^3 + c_{14}x^2y + c_{15}x^2z + c_{16}xy^2 + \\ & c_{17}xz^2 + c_{18}y^2z + c_{19}yz^2 + c_{20}xyz \end{aligned} \tag{5}$$

This equation is a cubic polynomial and is complete through the cubic terms of the 3D power series, therefore it has all the terms shown in (1).

The interpolation equation for the quadratic brick, with 20 nodes (interior nodes located at the center of its edges), is given in Equation 6; it is a fourth order polynomial in x, y and z.

$$\begin{aligned} f(x,y,z) = \quad & c_1 + c_2x + c_3y + c_4z + c_5x^2 + c_6y^2 + c_7z^2 + \\ & c_8xy + c_9xz + c_{10}yz + c_{11}x^2y + c_{12}x^2z + c_{13}xy^2 + c_{14}xz^2 + \\ & c_{15}y^2z + c_{16}yz^2 + c_{17}xyz + c_{18}x^2yz + c_{19}xy^2z + c_{20}xyz^2. \end{aligned} \tag{6}$$

4 Overview of the HIAC System

The HIAC volume rendering system for unstructured meshes uses software rendering based on the cell projection method and utilizes the absorption plus emission volume density optical model [27]. Either the Williams and Max [60] or the Wilhelms and Van Gelder [55] treatment of glow energy may be specified for use. A faster but less accurate renderer using hardware rendering is described later in this section.

The software system reads in an image specification file as defined in [61] (examples of which are given in Appendix B), generates the specified volume rendered image in software, and then writes to disk either an image file in SGI *RGBA* format or optionally as separate floating point R, G, B, and A files. The hardware version in addition writes the images directly through the graphics pipeline to the user's screen. Transfer functions for color and opacity are specified in separate files in a piecewise linear method as described in [61], examples of which are given in Appendix B. The radiance integration along a ray may be specified so as to use exact integration [60], which is appropriate when the cells are linear tetrahedra, or five point Gaussian integration which is appropriate for quadratic tetrahedra. For other zoo elements, the color and opacity are interpolated linearly along the ray segment inside the element as described in Section 5.3.

A faster, but somewhat less accurate method, which we call the *approximate method* and which is also done in software, assumes the opacity varies linearly along the ray segment and assumes the color is constant, equal to the average of the color at the front and the back of the ray segment. This is not exactly correct since the opacity along the ray segment hides the far color more than the near one, but the radiance integral is much quicker to evaluate.

In both the high accuracy mode and the approximate mode, the radiance integral is evaluated along a ray to each pixel, for every pixel that a cell projects onto. So if, on average, a cell covers 100 pixels, and there are 1,000,000 cells, then 100,000,000 ray integrations are performed in software.

The data ranges on which the transfer functions are actually linear are separated by data values which we call *breakpoints*. For the exact integration and the approximate methods, a cell is sliced into slabs at each transfer function breakpoint that occurs within a cell; in addition, cells are sliced at each user-specified isosurface value. For quadratic tetrahedra, the cell is sliced conceptually at all breakpoints and contour surfaces as a part of the integration procedure. This ensures that the color and extinction coefficient are smooth polynomials within a slab which is necessary to assure that the Gauss integration is exact. Within a slab from a linear tetrahedron, we can linearly interpolate either the color and extinction coefficient, or the scalar field. It would not be correct to interpolate the color or extinction coefficient if the cell contained a breakpoint in a transfer function.

For linear tetrahedra, if the color or density is linear throughout the cell, i.e. no breakpoints nor isosurfaces occur within the cell, then the cell is rendered as a whole. For nontetrahedral linear cells, including hexahedral cells from a curvilinear grid, if a breakpoint or contour value occurs within a cell, that cell is first tetrahedralized and then processed as described above, slicing the resulting tetrahedra as necessary. The system will also deal with quadratic tetrahedra, which it slices conceptually during the integration process whenever a contour lies within the cell.

Images may be generated in either perspective or orthographic projection, with any specified view transform and to any resolution. In the software version, near and far clipping planes parallel to the screen may be specified, in what we call *z-clipping*, in order to select a volume slab of interest, but this is not yet implemented in the hardware version (see Section 6). Any number of illuminated Phong-shaded semitransparent colored isosurfaces may be spec-

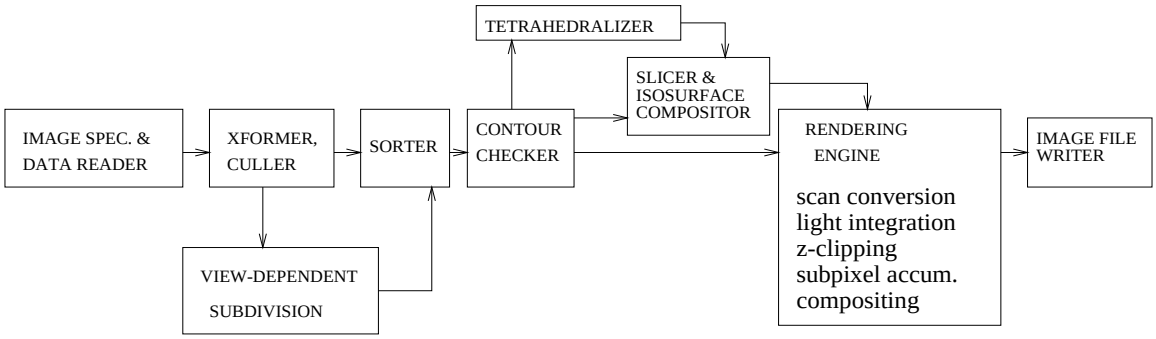

Figure 2: Schematic diagram of the HIAC volume rendering system. If cells have neither transfer function breakpoints nor contour surfaces in their interior, then the cells go directly from the sorter to the rendering engine. Otherwise, the cells are sliced into slabs, with nontetrahedral cells first being partitioned into tetrahedra. Quadratic tetrahedra go directly from the sorter to the rendering engine as described in Section 5.2.

ified for inclusion in the volume rendering. For linear tetrahedra, the contour surfaces are polygonal, and the surface normal for each polygon is used to shade the surfaces. We do not smooth-shade the contour surfaces because this might obscure important information in the visualization that is useful for determining whether the mesh has been properly refined. For higher order cells, the isosurfaces are created and shaded on a pixel by pixel basis, and not as a set of polygons, therefore these contours are smoothly curved surfaces within each cell.

Subpixel splatting may be specified, so that contributions from small cells that fall between pixel centers are included in the image. Any background color can be specified, as well any one of a selection of background patterns.

Fast previewing of images is facilitated by the use of graphics hardware through hooks to Williams' hardware-assisted cell projection system [59] based on Shirley and Tuchman's rendering algorithm [46] and Williams' Meshed Polyhedra Visibility Ordering (MPVO) algorithm [58], and an extension of the techniques of [46] to arbitrary convex polyhedra. These algorithms are described in Section 6. We currently offer two sorting algorithms, the exact but slower SXMPVO algorithm described in Section 8, and the faster by approximate MPVONC method described in [59, 58].

The HIAC system will also sort and render curvilinear data, provided the faces of the cells are not highly curved or severely deformed (crumpled, see Section 11.2). In the finite element method, cells can be deformed by a parametric mapping function, in which case the facets of the cells can be highly curved. How to volume render these nonconvex cells with curved facets is an important and interesting open question, as is the question of how to deal with crumpled cells.

A schematic diagram of the system is shown in Fig. 2. After the image specification file is parsed, the data set is read in, the view transform and the perspective transform (if applicable) are applied, and cells which project outside the screen window are culled. If a view-dependent subdivision is to be performed, then it is done next. Then the cells are sorted in visibility order from back to front, sliced (if necessary) into slabs bounded by contour levels

and transfer function breakpoints, and then the slabs (or cells) are scan converted and the results of the ray integration through the slab (or cell) for each pixel are composited into the image buffers. The image buffers hold the red, green, blue, alpha and z values in floating point format. The z buffer is used as a witness to verify the correctness of the visibility ordering. The alpha buffer is used to permit post-processing accumulation of more than one semitransparent image. The rendering engine does the scan conversion, radiance integration, subpixel splatting, z-clipping and compositing.

5 The Rendering Engine

Max [27] describes several theoretical optical models for light interacting with a volume density, each with differing degrees of realism. A volume rendering system can be created based on any one of these models. If the system is constructed faithfully according to its model, without the use of approximations, then that system will create accurate images. (If the differential equation for radiance can only be solved by numerical methods, then the system will create images to some predetermined degree of precision.)

A volume rendering system can either integrate the radiance over rays cast out from each pixel through the entire volume density, or project each cell in the volume density onto the screen in visibility order and integrate the radiance over each projected cell for each pixel covered by it. When the cell projection approach is used, a visibility ordering of the cells is required in order to composite the semitransparent volume cells into the image in back-to-front order.

The HIAC system, which uses the cell projection method, is based on the *absorption plus emission* optical model [27, 60], also known as the isotropic *density emitter* optical model of Sabella [42], for the volume effects. In this model, every point in the cloud absorbs light and also emits light (glows). The differential equation for the radiance along a ray towards the eye through the volume is:

$$\frac{dI(t)}{dt} = g(t) - \tau(t)I(t) \tag{7}$$

where t is a length parameter along the ray, and $I(t)$ is the radiance at t. The optical density or extinction coefficient of the volume at t, $\tau(t)$, is considered to be a physical property of each point in the cloud, and defines the rate that light is absorbed or occluded at that point.

The remaining term $g(t)$ is the glow energy emitted at each point of the cloud. There are two ways to treat the glow energy. Wilhelms and Van Gelder [55] treat the glow energy as a physical property of the cloud, whereas, Williams and Max [60] consider the glow energy to be defined as $g(t) = \kappa(t)\tau(t)$, where the chromaticity $\kappa(t)$ is considered to be a physical property of each point in the cloud. The HIAC system will generate images using either treatment of the glow energy, as chosen by the user.

We assume the use of piecewise linear transfer functions for specifying the dependence of chromaticity (or glow energy) and optical density on the scalar field being visualized.

By the use of an integrating factor and by applying boundary conditions at t_1 and t_2, we get the following integral equation for the radiance using the Williams and Max treatment of glow energy, see [27, 60]:

$$I(t_2) = I(t_1)e^{-\int_{t_1}^{t_2} \tau(t)\,dt} + \int_{t_1}^{t_2} e^{-\int_{t}^{t_2} \tau(u)\,du} \kappa(t)\tau(t)\,dt. \tag{8}$$

This equation is instantiated once for each of the three component wavelengths of light. The second term represents the glow energy along the ray segment, attenuated by the opacity in front of it, and the first term represents the incoming illumination $I(t_1)$ at the far end of the ray, also attenuated by the intervening opacity.

For Wilhelms and Van Gelder's [55] *neon and smog* treatment of glow energy, we get the following integral equation whose terms can be understood in the same way:

$$I(t_2) = I(t_1)e^{-\int_{t_1}^{t_2} \tau(t)\,dt} + \int_{t_1}^{t_2} e^{-\int_{t}^{t_2} \tau(u)\,du} g(t)\,dt. \tag{9}$$

The integral which is the second term on the right side of (8) and (9) can not be solved in closed form for general $\tau(t)$. However, if the scalar field and the transfer functions vary piecewise linearly along a ray segment within a cell, then the equations can be integrated exactly over each linear region. This solution is described by Williams and Max in [60] and discussed further in the next section together with its implementation.

More generally, let the second term of the right hand side (of either equation) be described as:

$$\int_{t_1}^{t_2} e^{a(t)} c(t)\,dt. \tag{10}$$

A general closed form solution for this integral is not known when $a(t)$ is cubic or higher order, regardless of the form of $c(t)$. Therefore, even though the $c(t)$ term is lower order in the neon and smog treatment, the integral still can not be solved exactly when the scalar field is quadratic or higher order (even with linear transfer functions). The neon and smog treatment does however permit the glow energy to be mapped independently to a different scalar field than the optical density, which is not possible with the other treatment. Nevertheless, we have created successful visualizations where κ and τ each depended on separate scalar fields using the Williams and Max treatment of glow energy. The main advantage of the Williams and Max treatment is that it makes the specification of the transfer functions somewhat more intuitive. For example, increasing the extinction coefficient makes the surface color more dominant, rather than making the image darker and ultimately black as is the case with the neon and smog model. More details on this and on the relative merits of the two different treatments of glow energy are given in an Appendix to [61]. The HIAC system allows the optical density to be mapped to a different scalar field than the color, and the contour surfaces can be keyed to a third scalar field.

The HIAC system uses the cell projection approach, therefore a visibility ordering of the cells is required. The sorting algorithm originally used in [26] was restricted to rectilinear volumes or Delaunay triangulations in 3D. Earlier versions of HIAC used the exact sorting

algorithm developed by Stein [50, 62] and the *Binary Search Partition–Exact Meshed Polyhedron Visibility Ordering* (BSP-XMPVO) algorithm [4]. The current HIAC system uses a different sorting algorithm, an efficient and exact algorithm developed at LLNL by Cook and Max specifically for HIAC, called the SXMPVO visibility ordering algorithm, which is described in [5] and in Section 8. The SXMPVO algorithm will handle nonconvex and disconnected meshes, and the cells may have sliding interfaces. However, the cells are expected to be nonintersecting, and the mesh acyclic. The SXMPVO is faster than either of its two predecessor sorting algorithms.

At present, the tetrahedron and brick, in their linear and quadratic forms, dominate practical applications [24]. Therefore, in this report, we limit our coverage of accurate volume rendering methods to these cells, and the prism. Similar techniques to those given in this report can be applied to other types of cells provided the interpolation function is fourth degree or lower. We have implemented the high accuracy volume rendering methods for the 4-node tetrahedron and 10-node tetrahedron, and we describe below how one might extend the system to deal with the linear brick and prism, and the quadratic brick. At present, the HIAC system will also handle bricks, prisms and pyramids, although not with the highest precision.

The next section describes how the HIAC software rendering engine processes linear cells. Section 5.2 describes the treatment of higher order cells; Section 5.3 deals with zoo elements; and, Section 5.4 describes how HIAC deals with elements with nonplanar facets. Finally, Section 5.5 describes the subpixel splatting procedure. The hardware rendering process is described later, in Section 6.

5.1 Linear Elements

Linear cells, as discussed in Section 3, are convex polyhedra where the scalar field is specified at the conventional vertices and varies linearly along the edges of the cells.

5.1.1 Linear Tetrahedral Elements

After visibility ordering the cells, each cell is checked to determine if the range of the scalar field within it includes any transfer function breakpoint values. If any are found, and the cell is tetrahedral, the cell is sliced at each breakpoint resulting in *slabs* in which the color and opacity are linear. Each slice is defined by a contour surface for the field value corresponding to a transfer function breakpoint. Since the scalar field varies linearly within a tetrahedron, the slices are planar and parallel. An example slab is shown in Fig. 3. If an isosurface is to be separately rendered, the tetrahedron must also be sliced at these contour values so the slabs and surface polygons can be composited individually in the correct order. Currently, if the cell is nontetrahedral, we first subdivide the cell into tetrahedra and then slice the tetrahedra into slabs.

The visibility ordering of the tetrahedra within a hexahedron, or the slabs within a tetra-

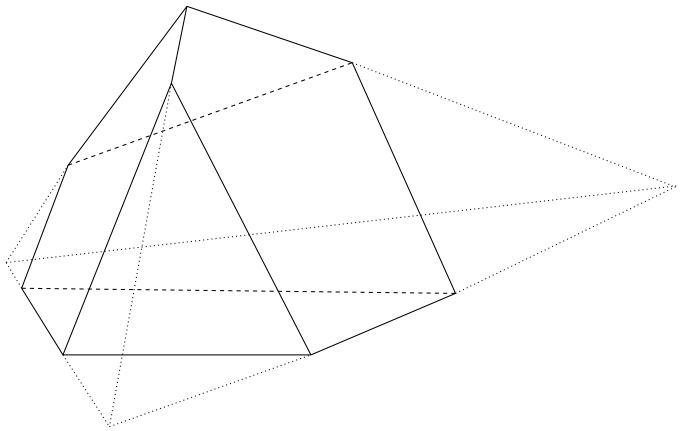

Figure 3: Example of slab of linear tetrahedron. Within the slab, the color and opacity are linear. The slicing planes defining the slab are contour surfaces for field values corresponding to transfer function breakpoints or for user-specified isosurface values.

hedron, is simple, and is done separately from the global visibility ordering of all the cells. (Methods like the marching cubes algorithm of Lorensen and Cline [21] can slice hexahedral cells directly, but the slices, which are curved surfaces, must be divided into triangles. If there are multiple contour levels inside a single cell, it is not clear that they will be nonintersecting, or that the volume pieces they slice off will be convex, as required by our projection algorithm.) We do not subdivide nontetrahedral cells into tetrahedra when the range of scalar function within the cell does not include a slicing value unless they would cause a sorting problem (see Section 5.4). Both the sorter and the scan converter will handle the different types of linear cells described in Section 1. The rendering engine will correctly process convex cells or slabs with any number of faces and vertices.

The slicing algorithm is robust, permitting up to three vertices of a tetrahedron to take on the same slicing value. When all four vertices have the same contour value, the set of points taking on the contour value is no longer a surface but contains the whole tetrahedron. In that case, the contribution of such a tetrahedron to the isosurface is neglected, leaving a hole in the contour surface, but the tetrahedron will still contribute to the volume rendering. When a contour surface intersects three vertices of a tetrahedron, the surface corresponds to one of the cell's faces, which may be shared with another cell. In that case it is important to render the contour polygon only once. The boundary polygons for the slabs are found from a case by case analysis of the ways a slice plane can intersect a tetrahedron, and the ways in which the slab between two consecutive slice planes can intersect the tetrahedron's face triangles.

For orthogonal projection, the back-to-front sorting order of the slabs within a tetrahedral cell can be determined from the z-component of the gradient of the scalar field S on the cell: $S(x, y, z) = c_1 + c_2x + c_3y + c_4z$. (Since the field is known at the four vertices of the cell, the four constants can be determined for the cell.) If the z-component of the gradient is greater than zero, then the back-to-front order starts with the slab having the largest scalar value.

For perspective projection, consider the slicing planes defining the slabs to be infinite parallel planes in world coordinates. If the viewpoint lies between two of these infinite planes which define a slab, we call that slab the *eye slab*. The slabs and contour surfaces are then composited in two groups: from one side up to but not including the eye slab, and then from the other side, up to and including the eye slab. If there is no eye slab, only one group is required, as in the parallel projection case.

Next the cells or slabs are sent in visibility order to the rendering engine for projection and accumulation. For the software implementation, the first step in this process is to scan convert the cell or slab. The front facing polygons bounding the cell are scan converted into a front z-buffer, with values z_f, and the back-facing polygons into a back z-buffer, with values z_b. The κ and τ values are bilinearly interpolated along edges and across scan lines, as in Gouraud shading, and saved in the front or back buffers, as κ_f and τ_f, or κ_b and τ_b, respectively. Then, for each pixel in the projection of the cell, the length l of the ray segment is computed as $z_b - z_f$, and the values of κ and τ are assumed to vary linearly between their values in the front and back buffers.

In parallel projection, this results in piecewise trilinear interpolation, where the subdivision into trilinear pieces depends on the projection of the polyhedron (see Section 5.3. Thus, as in piecewise bilinear Gouraud shading, the interpolation scheme is not rotationally invariant. However, for linear tetrahedra, or for slabs cut from them on which κ and τ are linear, this trilinear interpolation reduces to linear interpolation, which is rotationally invariant.

Next, the ray integration is performed for each pixel covered by the cell or slab. When the cells are linear tetrahedra and the transfer functions are piecewise linear, the integral in (10) can be integrated exactly as shown in [60] by completing the square of the exponent and repeated application of integration by parts, yielding:

$$\begin{aligned} I(t_2) = \quad & \frac{\alpha+\beta t_2}{\delta^2} - \frac{\mu+\nu t_1}{\delta^2} e^{(t_1-t_2)(2\gamma+\delta t_1+\delta t_2)/2} + \\ & \eta\delta^{-2.5} e^{-(\gamma+\delta t_2)^2/2\delta} \left(erfi\left(\frac{\gamma+\delta t_2}{\sqrt{2\delta}}\right) - erfi\left(\frac{\gamma+\delta t_1}{\sqrt{2\delta}}\right)\right) + \\ & I(t_1) e^{-(\gamma t_2 + \frac{\delta t_2^2}{2} - \gamma t_1 - \frac{\delta t_1^2}{2})} \end{aligned} \tag{11}$$

where, α, β, δ, μ, ν, γ, and η are functions of: the four constants c_i in the tetrahedral interpolation function $S(x,y,z) = c_1 + c_2 x + c_3 y + c_4 z$, the constants describing the applicable linear pieces of the transfer functions, for example $\tau(x,y,z) = a + bS(x,y,z)$, and the three ray parameterization functions, $x = u_1 + u_2 t$, etc., as described in [60]. The $erfi()$ function will be discussed below. Key terms in (11) are: δ since it appears in the denominator of several terms, and $\gamma + \delta t$, the numerator in the argument to $erfi()$. The term δ is equal to the slope of the pertinent piece of the optical density transfer function, i.e. b in the example above, times the slope of the (linear) scalar field along the ray. The term $\gamma + \delta t$ is exactly $\tau(t)$ as can be seen from the detailed derivation given in [60].

The *complex error function*, $erf()$, is defined as: $erf(z) = \frac{2}{\sqrt{\pi}} \int_0^z e^{-u^2}\, du$. The *imaginary error function*, $erfi()$, which appears in (11), is defined as $erfi(z) = erf(iz)/i$. In (11), $erfi$'s argument is either real when $\delta > 0$, or pure imaginary when $\delta < 0$. When its argument

is real, $erfi(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{u^2}\, du$. When its argument is pure imaginary, of the form $z = ib$, with real b, $erfi(ib) = i\frac{2}{\sqrt{\pi}} \int_0^b e^{-u^2}\, du$. When $\delta < 0$, the i in the latter expression cancels the i in the factor $\delta^{-2.5}$ in (11). When $\delta = 0$, the solution to the integral takes a different and relatively simple form as shown in [60], involving only the exponential function, and not $erfi()$. (The formula for this case in [60] can be further simplified by noting that when $\delta = q_2 = 0$, q_5 is also zero, eliminating half the terms.)

Originally in [60], we implemented the $erfi()$ functions in terms of the indefinite integrals $\int_0^x e^{-t^2} dt$ and $\int_0^x e^{t^2} dt$. We precomputed these integrals incrementally at equally spaced values of x using Simpson's rule, and stored the results in two tables. To evaluate (11), we interpolated values from these precomputed tables.

By introducing Dawson's integral, which is defined in [39] as $D(x) = e^{-x^2} \int_0^x e^{t^2} dt$, the solution to the integral equation can be simplified to eliminate the use of tables. Dawson's integral is related to the complex error function by:

$$D(x) = \frac{-i\sqrt{\pi}}{2} e^{-x^2} erf(ix) = \frac{\sqrt{\pi}}{2} e^{-x^2} erfi(x). \tag{12}$$

An efficient and accurate numerical approximation for Dawson's integral due to Rybicki [41] and described in [39] enables the calculation of $erfi()$ without the use of tables. The accuracy of Rybicki's approximation increases *exponentially* as the step size, h, used in the approximation, gets small. We use $h = 0.4$ which gives an accuracy of about 2×10^{-7}. The function $erfi(x)$ for imaginary x can be reduced, by a trivial change of variables, to the Error integral, the integral of a Gaussian normal distribution, for which subroutines also exist, as described in [39]. Appendix A discusses details of implementation and how to avoid overflow in the exponentials.

For linear tetrahedra, the HIAC system uses the exact solution from [60], utilizing subroutines for Dawson's integral and the Error integral as described above.

When Wilhelm and Van Gelder's neon and smog treatment of the glow energy is used, the techniques given above still apply, but the term $\kappa(t)\tau(t)$ in the integrand is replaced by $g(t)$ which is now linear rather than quadratic for linear scalar data. Unfortunately, this does not permit any significant further simplification in the calculus.

When a perspective view is specified, a bit of care is required to do mathematically correct interpolation and integration, since the distance metric along an edge or ray is distorted by the perspective transform. After performing the perspective transform, the scalar field no longer varies linearly along the edges of a cell nor on a ray through a cell. (For example, the midpoint of an edge in parallel projection is no longer the midpoint of that edge after the perspective transform.) Therefore integration techniques that are suitable for linear functions no longer pertain. Our approach to this problem is to reverse the perspective transform and do the interpolation and integration in world coordinates rather than screen coordinates. When this is done, the length of the ray segment must be computed as a 3D (slanted) distance, rather than just a difference in z values. (For a perspective ray from the origin through a pixel at $(\overline{x}, \overline{y}, 1)$, the difference of the world coordinate z-values of the endpoints of the ray segment must be multiplied by $\sqrt{\overline{x}^2 + \overline{y}^2 + 1}$.) The details of this

calculation are tedious and the interested reader is referred to our code.

Near and far clipping planes parallel to the screen may be specified to achieve a volume slab of interest. This z-clipping is accomplished as follows. Cells entirely in front of the near clipping plane are skipped, as are cells entirely behind the far clipping plane. Cells intersecting the slab are processed normally, but for each pixel, the viewing-ray/cell intersection segment is restricted to the region inside the slab. This could be done efficiently by 3-D polyhedron clipping, but the above per pixel *scissoring* alternative was easier to code. The z-clipping is also implemented for quadratic cells.

5.1.2 Other Linear Elements

The interpolation equations for the other linear elements, the brick (see Equation 2) and the prism (see Equation 3) are not linear therefore the methods described above do not pertain. Instead we must use the techniques we will develop in the next section for the quadratic tetrahedra. Therefore we postpone treatment of the other linear elements until Section 5.2.2.

5.2 Higher Order Elements

As discussed in Section 3, quadratic cells are cells where the scalar field varies quadratically along the edges of the cell. In this section, we deal in depth with quadratic tetrahedra which have six interior nodes, one per edge, as in Fig. 1. The other higher order elements can be dealt with in a similar manner and are discussed briefly in Section 5.2.2, as are the linear brick and the linear prism which have nonlinear interpolation functions.

5.2.1 Quadratic Tetrahedral Elements

In a quadratic tetrahedron, the scalar field varies quadratically along any ray segment through the cell because (4) contains only quadratic terms. Inside these cells, contour surfaces will be curved, therefore the slabs will be curved, and a viewing ray may intersect a single slab twice. Because of this, we do not actually partition the cells into slabs as we did for linear tetrahedra, but rather process each ray through a cell in segments. In each ray segment the color and optical density vary smoothly.

The interpolation function for a quadratic tetrahedron has the form of (4), which we repeat below:

$$f(x, y, z) = c_1 + c_2x + c_3y + c_4z + c_5x^2 + c_6xy + c_7y^2 + c_8yz + c_9z^2 + c_{10}xz.$$

Substituting the coordinates of the ten nodes, along with their field values, into this equation, we get ten equations for the ten unknown polynomial coefficients c_i, which we solve with the LINPACK linear algebra package.

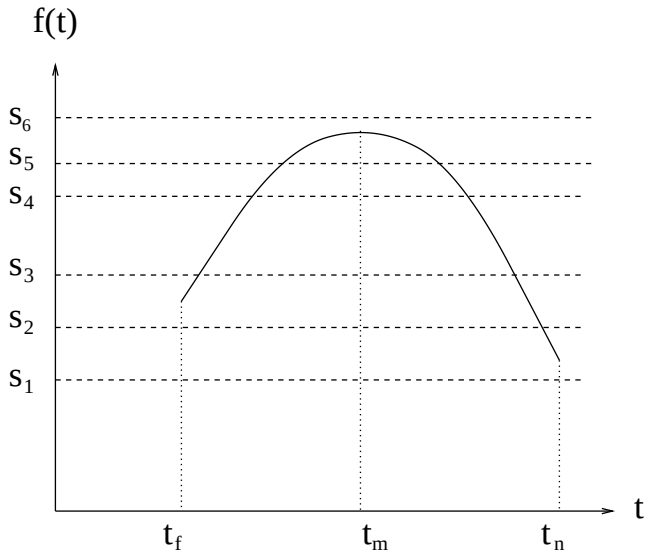

Figure 4: The variation of the scalar field $f(t) = at^2 + bt + c$ along a ray, towards the eye, through a quadratic tetrahedron as a function of the ray parameter t. Example regions defined by contour surfaces corresponding to transfer function breakpoints are also shown. The horizontal regions between s_i and s_{i+1}, for $1 \leq i \leq 5$ correspond to curved slabs in the tetrahedron. The ray enters the tetrahedron at t_f.

The segment end points are found as follows. The ray equations parametrized by t, are $x(t) = -\overline{x}t$, $y(t) = -\overline{y}t$, $z(t) = -\overline{z}t$. (We assume the viewpoint is at the origin, the pixel is located at $(\overline{x}, \overline{y}, \overline{z})$, where $\overline{z}$ is the distance from the viewpoint to the screen, and the ray is in the direction of light flow, with t increasing towards the eye.) Substituting the ray equations into the quadratic interpolating function shown in (4), gives a quadratic polynomial in one variable: $f(t) = at^2 + bt + c$. When a is nonzero, this polynomial will take on a maximum or minimum at a single t value $t_m = -b/(2a)$. If the ray segment does not contain t_m, the quadratic polynomial is monotonic in the segment. Otherwise, it contains both increasing and decreasing regions, as shown in Fig. 4. These regions, and their monotonic direction, can be determined from the sign of a, which is the sign of the second derivative of f, and the location of t_m relative to t_n and t_f, the near and far endpoints of the ray segment.

For every relevant slicing value, s_i, the corresponding breakpoints in t can be found by using the quadratic formula for the roots of $f(t) = s_i$. Between every pair of consecutive breakpoints s_i and s_{i+1}, the color and optical density will each be represented by a smoothly varying polynomial in t. The points in the quadratic tetrahedron, which have scalar values in the interval $[s_i, s_{i+1}]$, define a curved slab bounded by the contour surfaces at the break points and parts of the tetrahedron's surface facets.

We will refer to this as the slab $[s_i, s_{i+1}]$. The ray/slab intersection segments and their order along the ray can be determined by comparing $f(t_n)$, $f(t_f)$, and $f(t_m)$ with the slicing values s_i. In the case illustrated in Fig. 4, $(s_1 < f(t_n) < s_2 < f(t_f) < s_3 < s_4 < s_5 < f(t_m) < s_6)$. Therefore, the ray enters the slab $[s_2, s_3]$ through a tetrahedron face at the left where f is increasing, passes through the slab $[s_3, s_4]$, continues through slab $[s_4, s_5]$, then enters the slab $[s_5, s_6]$, continues through $[s_5, s_4]$ (with f decreasing), and then slabs $[s_4, s_3]$, $[s_3, s_2]$, and

$[s_1, s_2]$, and finally exits through a face of the tetrahedron. Though t_m is shown in Fig. 4, it is not one of the breakpoints; f reaches its maximum on the ray segment inside the slab $[s_5, s_6]$, but continues smoothly past its maximum, as do $g(t)$ or $\kappa(t)$, and $\tau(t)$, so there is no need to subdivide the integration there. Other cases can be handled similarly: the code has two loops over the increasing and the decreasing ranges of $f(t)$, but one may not be needed.

Assuming the Williams and Max treatment of glow energy, let $\kappa(t)$ and $\tau(t)$ be the chromaticity and optical density, respectively, at position t along the ray. Then the total opacity from a ray segment $[p, q]$ is:

$$e^{-\int_p^q \tau(t)\,ds} \tag{13}$$

and the total radiance or color added by that segment is:

$$\int_p^q e^{-\int_t^q \tau(u)du} \kappa(t)\tau(t)\,dt. \tag{14}$$

For $\kappa(t)$ and $\tau(t)$ quadratic in t, (13) can be integrated exactly, but numerical integration is required for (14), because the $a(t)$ in (10) is a cubic polynomial. We have used five point Gaussian integration [39], which gives exact answers for polynomials of up to degree nine, and very good approximations for sufficiently smooth functions that are well approximated by such polynomials, but poor approximations for functions which are not smooth. This is the reason for breaking the range of integration up into the subsegments where $\kappa(t)$ and $\tau(t)$ are smooth polynomials. The color (14), and opacity (13) on these subsegments are composited in the back-to-front order described above.

If a semitransparent contour surface is requested at the near breakpoint of a subsegment, it is composited after the subsegment. The surface normal is computed from the partial derivatives of (4), and used for Phong shading, as well as to make the surface appear more opaque when it is seen edge on, as if it were a finite thickness of partially absorbing glass. This makes the contour surfaces appear appropriately curved within a cell even in the absence of reflected light. By Equation 4, the contour will be a smooth quadric surface within each cell. However, finite element simulations rarely produce results which are C^1 across cell boundaries, so the contour surfaces may not be globally smooth.

5.2.2 Other Elements with Nonlinear Interpolation Functions

The interpolation function for the linear brick is given in Equation 2. It is trilinear, therefore contours within the cells are curved and so the methods described above pertain. The function $f(t)$ is cubic because of the xyz term in Equation 2. To find the roots of the cubic polynomial, $f(t) - s_i = 0$ we can use the closed form solution given in [68].

The interpolation function for the linear prism, given in Equation 3, has the bilinear terms xz and yz, and so contour surfaces in the interior of the linear prism will be curved. Substituting the ray parameterization into the interpolation equation, we get a quadratic polynomial which can be analyzed on a case by case basis similarly to that used for the quadratic tetrahedron.

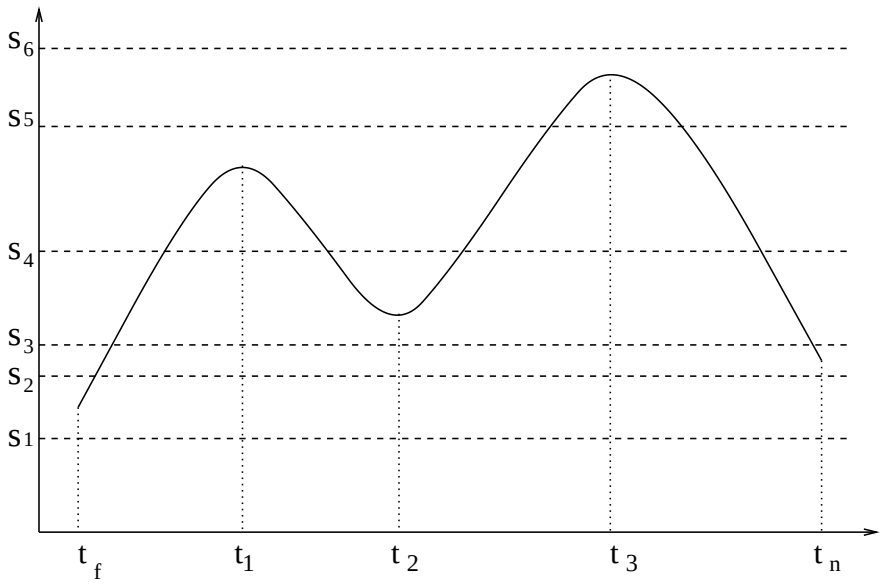

Figure 5: Analysis of ray segment through a quadratic brick element.

The interpolation equation for the quadratic brick, given in Equation 6, is a fourth order polynomial in x, y and z, so its evaluation along a linear ray gives a fourth order polynomial $f(t)$ in the ray parameter t. The points on a ray where the polynomial takes on a contour value s_i can be found analytically by the closed form noniterative solution of the quartic equation first published by Ferarro, see [63]. Here a case analysis, similar to but more complex than the one described above for quadratic tetrahedra, is required to find the regions where the polynomials $\tau(t)$ and $g(t)$ are smoothly varying, *i.e.* include no breakpoints. A diagram of the case analysis for finding the ray segments is given in Figure 5. The points t_1, t_2, and t_3, where $f'(t) = 0$ separate the monotone ranges of $f(t)$, can be found as roots of the cubic polynomial $f'(t)$. Either Gaussian quadrature or the power series method of Novins and Arvo [36] can be used to do the integration.

The interpolation equation for the cubic tetrahedron, given in Equation 5, is of degree three, hence the field varies cubically along any ray through the cell; therefore it can be dealt with in a similar way to that described above for the other higher order elements.

5.3 Treatment of Zoo Elements

First we take up the treatment of zoo elements with planar facets, and then in Section 5.4 we discuss how deformed elements with nonplanar facets are dealt with. In practice zoo meshes often contain deformed elements so some details of our treatment of zoo meshes are postponed until Section 5.4.

At present the high accuracy treatment described above is only used for linear and quadratic tetrahedra, not the other linear zoo elements. In the interior of these elements, i.e. the

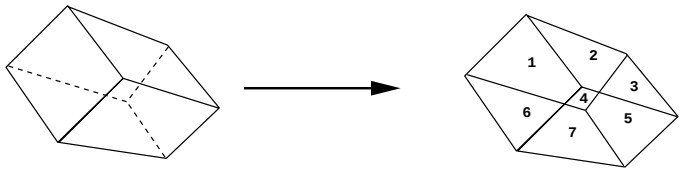

Figure 6: Example of the division of the screen projection of an entire brick into polygons (seven in this case) by the projections of the brick's edges.

linear brick, prism and pyramid, the scalar field varies non linearly, e.g. in a linear brick the scalar field varies trilinearly. In addition, isoparametric or superparametric elements may be deformed nonlinearly, so that the position in space varies nonlinearly with the coordinates of the standard element, i.e. a nonaffine transform is involved.

HIAC has several methods for dealing with nonlinearities in the scalar field. First, all elements can be divided into linear tetrahedra, giving a piecewise linear interpolation of both position and scalar field, which can be rendered accurately, but in general will not agree with the nonlinear scalar field in the interior of the undivided element. Alternatively, the element can be projected as a whole. This basically involves dividing the screen projection of the element by the projections of all its edges, resulting in a number of polygons, and then interpolating in software or hardware across these polygons. See Figure 6.

The high accuracy software polygon scan conversion method interpolates scalar values linearly along the projected edges to get values at the endpoints of horizontal scan line segments, and then linearly across each scan line segment. The integration along the ray segment through each pixel in a scan line then also assumes linear interpolation of the scalar field along the ray. The effect, in an orthogonal view, is trilinear interpolation within each polygon subtrapezoid where the scan line segments join the same two edges, and is thus piecewise trilinear in the whole element. It is continuous if every polygon in the projection is convex, but cases may arise for nonconvex elements where this is not the case, and the piecewise interpolation is discontinuous.

The hardware method is similar, but each polygon in the projection is subdivided into a triangle fan, so the result is piecewise linear across the image plane, coupled with linear along the viewing rays, giving a piecewise bilinear interpolation which is always continuous.

The technique of subdivision by element edges discussed in the paragraph before last will divide each element's projection independently, and may produce gaps or overlaps in the viewing ray segments. This technique is not consistent with subdividing all the quadrilateral faces into triangles and rendering the polygons in Figure 8, because a single polygon from Figure 7 will usually overlap several polygons from Figure 8. The projection of an interior face will be subdivided differently by the projected edges of each of the two cells it bounds, resulting in two different interpolations of z. This can produce a gap or overlap between the ray segments for these two adjacent cells, causing errors in the volume rendering. In most of our applications, the quadrilaterals are almost planar, and the errors introduced are not large.

We also have a slower face subdivision alternative which divides all quadrilaterals into two

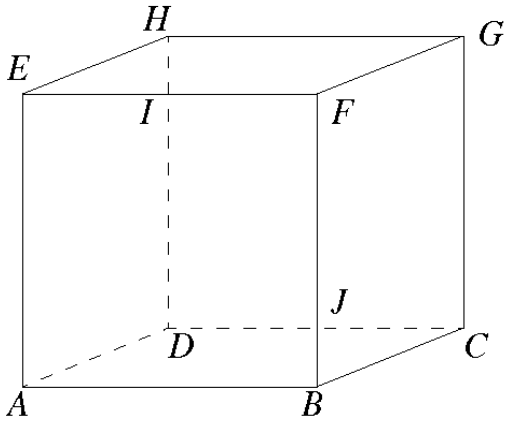

Figure 7: Projection of a hexahedron.

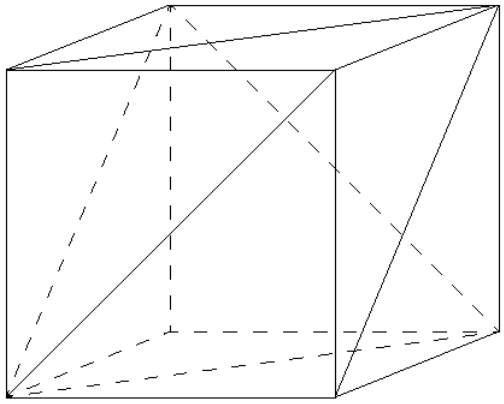

Figure 8: Hexahedron with all faces divided into triangles.

triangles, then the depth and scalar values will vary linearly across each triangle, and subsequent subdivision of these triangles by the projections of other edges will not change this. This will again give piecewise bilinear interpolation, but it will now respect the face subdivision. The subdivision into triangles is chosen consistently for the two elements sharing a quadrilateral face, so the position interpolation is continuous across the mesh. Therefore, no gaps or overlaps in viewing ray segments will occur, so this method is more accurate. This method will renders all the polygons of Figure 8 without z interpolation inconsistencies.

In a volume region where the scalar field is varying slowly, the subdivision of all faces into triangles will be more accurate because it cannot cause gaps or overlaps along the viewing rays. However, neither method agrees with the finite element interpolation functions, so it is difficult in general to say which is more accurate. Also, since triangles are always convex, the face subdivision method produces a continuous interpolation of the scalar field. However, it adds extra element edges and creates more polygons in the projection, and will thus be slower.

Note that the piecewise linear interpolation across the tetrahedral subdivision is view-independent. However, all the other interpolation methods are view-dependent because they use linear interpolation across viewing ray segment from the segment endpoints. As an element rotates, the endpoints of the ray segment through a given 3D point will move from one element face to another, and thus be interpolated using different vertex scalar field values, which will give different answers unless the scalar field is linear.

5.4 Deformed Elements

The HIAC system will sort and render cells with nonplanar faces, which occur in deformed and curvilinear meshes, provided the cells are not highly deformed, so that we can approximate a nonplanar quadrilateral face by two triangles when necessary. It does this by partitioning such cells into tetrahedra when necessary, depending on the viewpoint location. A mesh of zoo elements can always have its cells subdivided, as an abstract topological cell complex, into tetrahedra so that any quadrilateral face shared by two cells can be divided consistently into the same two triangles by the tetrahedral subdivisions of the two cells. For a proof see [28].

HIAC currently renders zoo elements, with the topology, but not the geometry, of tetrahedra, square pyramids, triangular prisms, and cubes, as read from files in the SILO format [47]. The SILO files have an array of numbered vertex positions, and for each of the four cell types, an array of cells, each defined by a list of indices into the vertex array. Thus elements are specified only by their vertex positions, and the cube topology may correspond to a hexahedron with nonplanar quadrilateral faces.

In our data structures, as in the SILO data, there is a large array of vertex coordinate triples. Each cell is defined by its type (as determined by the number of its vertices), and a list of vertex indices pointing into a global vertex array. Each cell type has a prespecified list of faces, defined by indices into the cell's list of vertices. Composing these two indexing

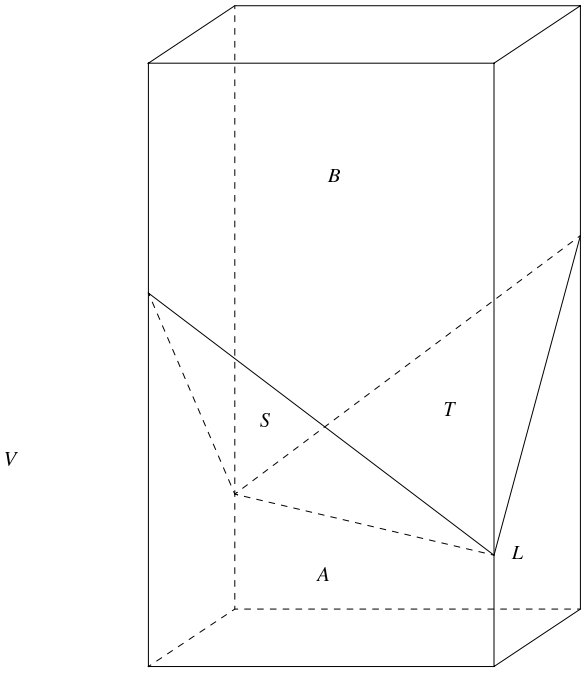

Figure 9: A nonplanar face between cells A and B, approximated by S and T.

functions then assigns to each face vertex a *global index* into the global array of vertices.

From certain viewpoints, nonplanar faces such as shown in Figure 9 can cause a problem for volume rendering and sorting since a ray may enter, exit, and then re-enter the same cell. For a cell A and a specific viewpoint V, consider a quadrilateral face F divided into two triangles S and T by a diagonal from the vertex L, as shown in Figure 9. A viewing ray from V can exit A through S and enter A again through T. This means cells A and B form a visibility cycle, since cell A is both in front of and behind cell B. This happens if and only if the outward normal to S has a positive dot product with the vector VL, the outward normal to T has a negative dot product with VL, and both V and the interior of triangle S lie on the same side of the plane of triangle T. In this case, we call F a *problem face*, and A a *problem cell*. Problem cells like A are subdivided into tetrahedra, to break the visibility cycle (see Section 5.4.2). Note that a particular face or cell is a problem face or cell only for certain viewpoints, so the subdivision into tetrahedra is view-dependent.

Typically, unstructured data sets, including FEM data sets, have no record of the shape functions used to define the element shape. So we do not know the shape of the nonplanar faces. A natural assumption is that they are hyperbolic paraboloids, resulting from bilinear interpolation between the four vertex positions as given by the finite element formulas of Section 3, restricted to to one of the quadrilateral faces.[1] But volume rendering using such

[1]This is so since these formulas when used to calculate the shape of the element are applied to the standard unit cube, pyramid, and prism. For example, the vertices of the standard unit cube are at $(0,0,0),(0,0,1),(0,1,0),...,(1,1,1)$. The formulas of Sections 3 are then used with these coordinates to give vector valued equations, i.e. for (x,y,z) at each point within the standard element. Along a face where one of the coordinates of the standard cube is fixed, the function of the other two coordinates is bilinear.

faces would require tracing rays to intersect the parametrized face surfaces, which is difficult in software and not possible in hardware-based cell projection.

Instead, we approximate the unknown shapes by piecewise linear interpolation. We divide the problem quadrilateral faces into two planar triangles, and extend this to the problem cells by slicing them into tetrahedra. If a quadrilateral face F separates cells A and B, we must make sure that it is divided by the same diagonal when considered as a face of each. We do this by starting the diagonal from the quadrilateral vertex which has the lowest global index in the vertex list for the face. In [28] (and earlier in another context in [35]) is a proof that if the face diagonals are chosen this way, each of the zoo elements can be subdivided into tetrahedra whose edges include the chosen diagonals. The proof starts with the pyramids, which are trivial, observes that a prism can be sliced into a tetrahedron plus a pyramid, and then shows that a cube can be sliced either into five tetrahedra or else into two prisms.

5.4.1 Data Structures for Zoo Elements

As mentioned above, the vertices are stored in a large array. The SILO data has no information on the connectivity of cells across common faces, so this must be reconstructed after the data is input. The data structure for each cell is:

```
struct NewCell {
  char subdivided;       // currently subdivided
  char oldsubdivided;    // subdivided in last frame
  char numbInbound;      // for directed graph sort
  char type;             // zoo element or new tetrahedron
  char actual;           // actual, virtual, or degenerate
  char cycleTestBit;     // for detecting visibility cycles
  char notVisited;       // for MPVONC sort
  char nverts;           // number of vertices
  int UsedTIndex;        // into list of cells actually used
  struct Newcell parent; // parent of tetrahedron in block or pointer
                         // to a block of tetrahedra for subdivided cell
  struct NewFace **face; // head of face list
  int vertex[4];         // vertex pointers
}
```

Extra space for more vertices is allocated if `nverts` is more than four. The list of face pointers is actually an array stored after the array of vertex pointers, but because `nverts` is variable, it must be accessed via a pointer.

The following data structure for the faces allows either one subface, for triangles, or two subfaces, for quadrilaterals.

For more information on this see [24], page 122.

```
struct NewFace {
  short concave;     // 1 if a problem face; 0 if not
  short nsubfaces;   // 1 for quadrilateral; 0 if not
  struct Subface[1]; // more space allocated if needed
}
```

Each subface is a triangle, and therefore has a linear plane equation. The subface points to the two cells that share it. The first entry points to the cell with the lowest index; a tetrahedron from a subdivision gets its parent's index. For exterior subfaces, the second pointer is -1.

```
struct Subface {
  struct Newcell *shared[2];
  float A;    //  The plane equation is:
  float B;    // Ax + By + Cz + D = 0.
  float C;
  float D;
  char arrow; // graph edge direction
}
```

The arrow's direction is with respect to the first shared cell, `shared[0]`. It indicates whether the viewpoint is on the same side of the plane of the triangle as cell `shared[0]`, on the opposite side, or exactly on the plane.

The plane equation is computed once when the geometry is determined, but the arrows must be recomputed each time the viewpoint moves. The plane equations and arrows for the two subfaces of a quadrilateral face determine whether it is a problem face (setting `concave`) and which of the cells it bounds is a problem cell (setting `subdivided`). See Section 5.4.2 for details. If `subdivided` is `true` and `oldsubdivided` is `false`, the cell is subdivided by taking a block from one of four preallocated arrays of such blocks, for pyramids (which need two tetrahedra), prisms (which need three tetrahedra), hexahedra requiring five tetrahedra, and hexahedra requiring six tetrahedra. (Actually, we currently use seven arrays instead of four, because of an earlier attempt to save time by reusing preset face pointers that point to internal faces inside the block, which required a separate list for each block topology.) When a cell is subdivided, the shared pointers originally pointing to the cell must be revised to point to the new tetrahedra. This is the reason for including the shared pointers in the subfaces; a quadrilateral face pointing to a single cell may later need to point to two subtetrahedra.

If `subdivided` is `false`, and `oldsubdivided` is `true`, the parent pointer in the `NewCell` structure is used to restore the pointers to their state prior to subdivision, and the block is placed on a free list for its topological type. Below is the data structure for a block of cells.

```
struct BlockOfCells {
```

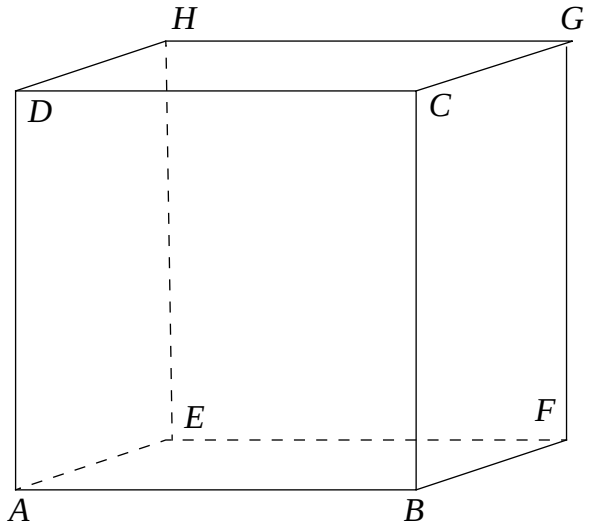

Figure 10: Projection of a cube.

```
    char type;                 // inherited from parent type
    char used;                 // 1 if currently in use; else 0
    short kind;                // which block topology
    struct BlockOfCells *next; // in lists of used & free blocks
    struct NewCell tets[2];    // more space allocated if needed
}
```

The internal faces are stored directly after the end of the `tets[]` array, in the memory allocated for the block, to make memory reference more local.

The data structures for data sets with only tetrahedra, either linear or quadratic, are much simpler than those for zoo elements and are described in Appendix C

5.4.2 View-Dependent Subdivision

Subdividing all hexahedral elements in a grid completely into tetrahedra could multiply the number of cells by as much as six, and make sorting and projection take longer. For example, the hexahedron shown in Figure 10 divides the image plane into 5 quadrilaterals and 2 triangles, giving 7 polygons with a total of 41 vertices (counting repetitions, as would be necessary in OpenGL polygon calls). Now suppose this cell is subdivided into the 6 tetrahedra shown in Figure 11. Four of these tetrahedra require 4 triangles for hardware rendering, and two of them require 3, giving 22 polygons with a total of 66 vertices. Although OpenGL triangle fans can reduce the latter vertex count, the basic polygon and geometry overhead is significantly larger for the subdivided case. The rasterization effort is also higher, since most pixels in the projection of the original hexahedron are covered by at least 3 of these triangles, and those in the projection of quadrilateral $ACGE$ are covered by 4.

Therefore, we subdivide only the cells which are intersected by a viewing ray in more than

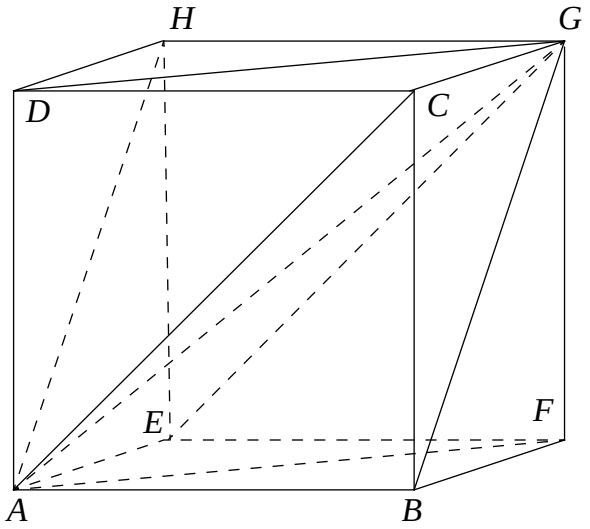

Figure 11: Cube divided into six tetrahedra.

one segment. We assume that the distortion is mild enough so that this can happen only when the viewing ray intersects a single curved face more than once. Our strategy is to think of all quadrilateral faces as in principle already divided into two triangles by the diagonal from their smallest vertex, but to actually project as quadrilaterals the nonproblem faces.

A problem cell such as A, shown in Figure 9, is temporarily[2] subdivided into tetrahedra. However, the cell B on the opposite side of F need not necessarily be subdivided. Instead, the quadrilateral face F of B is temporarily replaced by the triangles S and T for the purpose of sorting and projection from the current viewpoint. If B is later determined to also be a problem cell, due to a different problem face, the rest of B can be subdivided consistently, as shown in [29].

For each new viewpoint, all cells are examined to check whether they are problem cells, and if they are, the temporary changes discussed above are made. Then the visibility sort is run on the modified data structure.

We organized the data structure for the faces so that it can be easily updated if a problem cell is subdivided. Each quadrilateral face has two triangular subfaces, which are determined as soon as the mesh topology is known. Each subface data structure (see Section 5.4.1) has room for its plane equation, and for two cell pointers, to the two cells that share it (unless it is an exterior face, with only one such cell). If a problem cell is subdivided into tetrahedra, these cell pointers are revised to point to the new tetrahedra. The new tetrahedra have parent pointers to the cell they subdivide, to help in rerouting the cell pointers when a subdivided cell is restored to its original state. We intend to keep the subdivisions for the previous frame, and only add or delete the subdivision of cells whose problem state changes.

The data structures for the faces of the original cells are stored in two large linear arrays, one for triangles and one for quadrilaterals. These arrays are accessed in order when the

[2]A cell is subdivided only when the cell is a problem cell. When the viewpoint changes such that a problem cell becomes no longer a problem cell, that cell reverts back to its unsubdivided state.

new viewpoint is used with the plane equations to determine the graph edge directions, and which cells must be subdivided. This increases cache coherence.

The new tetrahedra for a subdivided cell, and their new separating faces, interior to the original cell, are stored together in a block of memory, to provide memory locality. There are four sizes of blocks, containing 2, 3, 5, or 6 tetrahedra, for the four subdivision cases discussed in Section 5.4.1. These are allocated in four large arrays, with lists of the used and unused blocks in each array. When a cell no longer needs subdivision, its block is put in the unused list, and can be reused.

In addition to the sorting problem mentioned above, there are other reasons to subdivide a cell into tetrahedra. As mentioned in Section 5.1.1 this is expeditious when an isosurface cuts a cell or where a breakpoint in the transfer function occurs within a cell.

We have found that the percentage of the total number of cells that must be divided can vary greatly depending on the viewpoint and on the degree of twist or deformation of the elements.

5.5 Subpixel Splatting

In curvilinear or adaptively refined unstructured meshes designed to concentrate small cells near shocks, boundaries, or other regions of rapid change or special interest, projections of tiny but important cells may fall between the pixel centers. This can also happen due to perspective foreshortening. Any volume rendering algorithm which samples the image only at pixel centers may therefore miss significant details entirely, or include them with an inappropriate weighting. This is the case in both ray tracing and cell projection methods.

The theoretically correct solution to this problem is to determine an analytic representation for the image as a function of the continuous coordinates on the image plane, and then convolve it with a presampling filter kernel, before sampling it at the pixel centers. Because of the geometric and analytic complexity of a volume rendered image, this is a formidable task.

We use an approximation to this analytic anti-aliasing, suggested by Westover's *splatting* technique [54]. If a cell's projection overlaps too few pixels (for example, less than two), we assume that its color and opacity effects on the image are concentrated at its center of gravity. We therefore take a delta function at the projection of the cell's center of gravity, and multiply it by the volume of the cell, the perspective projection area shrinkage factor, and the color and opacity at the cell's center of gravity. We then convolve this weighted delta function with a presampling filter kernel (described below), which is equivalent to taking a weighted translated copy of the kernel. The result is a splat to be composited onto the image.

If subpixel splatting is turned on, when the rendering engine gets a cell, we first do a rudimentary scan conversion to determine the number of pixels covered. If the pixel count is larger than a threshold, we repeat the scan conversion, doing the analytic integration

for color and opacity, and composite the result into the image. Otherwise we composite a weighted translated copy of the filter kernel.

We use a piecewise biquadratic kernel, the product of two identical 1D piecewise quadratic kernels in x and y, the B-spline kernel. This kernel is the twice iterated convolution of a pixel-sized box filter with itself. In spatial frequency, this filter has the Fourier transform $sin^3(\pi x)/(\pi x)^3$, which greatly attenuates frequencies greater than the Nyquist limit, and so gives good anti-aliasing. However it does cause some minor blurring, since frequencies less than the Nyquist limit are also attenuated, and the footprint of each cell is a 3 by 3 square of pixels. A wider filter, such as the one we use, is superior to a pixel sized box filter kernel when bright objects much smaller than a pixel move during animation. With a box filter kernel (area averaging), the bright object would suddenly jump from one pixel to an adjacent one when it crossed the edge between them, but with our wider kernel, the contributions smoothly fade up and down.

Rather than precompute and store a high resolution version of this splat, as Westover did, we just evaluate the simple quadratic polynomials each time they are needed. The polynomial variables are the fractional subpixel coordinates of the projection of the cell's center of gravity. The original algorithm of Westover used splats whose footprint decreased as the projected splats got closer together, but this method could also cause splats to be lost between pixels! Our solution is to keep the splat size to a three pixel square, and decrease the color/opacity amplitude instead, as described above. This is also the approach of Swan et al, [52]. Another approach (for regular grids only) is given by Mueller and Yagel in [32]. They use summed area tables to compute the integral of the splat footprint over the pixel area, so all splats will contribute their effects completely to the image.

We tested subpixel splatting by dividing a cube into a large number of tiny tetrahedra, each smaller than a pixel, and compositing their splats. The result was the same as the analytic integration over the projection of the five larger tetrahedra representing the original cube, except for slight blurring.

This splatting scheme is not a perfect solution to the antialiasing problem. Suppose a tiny cell is very bright but it is totally occluded by another tiny cell directly in front of it which happens to be dark and very opaque. First the tiny bright cell contributes a proportion to a nearby pixel, then the totally opaque cell contributes a proportion to the opacity, but overall the pixel will incorrectly retain some brightness. Variants of this problem with per pixel compositing occur with any scheme that does not represent the complete geometric projection of all cells overlapping the filter kernel. We are currently working on an analytic anti-aliasing scheme which does take into account the complete geometry, but we expect it to be very slow.

6 Hardware-Based Polyhedron Projection

The volume rendering system described to this point is not interactive, because it uses a precisely correct sorting, and a slow, accurate analytic or numerical integration along each ray

segment on which the transfer functions are linear. For rapid preview, polygon-based rendering hardware can be used instead. The sorting of the cells, the view-dependent subdivision in Section 5.4.2, and the subdivision of cells containing contour surfaces in Section 5.1.1, is performed as in the software version. Only the final scan conversion and ray integration steps are approximated in hardware.

Shirley and Tuchman [46] divided the projection of a tetrahedron into from one to four triangles, and used hardware scan conversion, transparency, and back-to-front compositing to produce an image. Stein et al. [50] point out that the linear transparency interpolation between the triangle vertices replaces what should be an exponential computation per pixel in (8), and can produce Mach bands. They suggest a more accurate method using hardware texture mapping, which we have now generalized from tetrahedra to arbitrary convex polyhedra.

In the discussion below, the opacity τ refers to the extinction coefficient, or differential opacity, as specified by the scalar value at a 3D point as indicated in Equation 7, Section 5, while the opacity α refers to the total integrated opacity along a ray segment, as used in the compositing. For τ constant on a ray segment of length l, $\alpha = 1 - e^{-\tau l}$. In this section, we will assume that the transfer functions are linear in s. In fact, our system supports piecewise linear transfer functions, by slicing cells into subcells inside which the transfer functions are linear, as discussed above.

The hardware-based method discussed below will be mathematically equivalent to analytic integration along viewing rays when:

a. The projection is orthogonal rather than perspective.

b. The faces of the cells are planar.

c. The scalar s varies linearly across cells, so that R, G, B, and τ are linear.

d. The color is constant per cell, and only τ varies linearly.

Condition (d) is stricter than that for software analytic ray segment integration per pixel, and arises because of limitations of the hardware rendering. As the discussion proceeds, we will explain the approximations we must make when one of these conditions is violated.

Figure 7 shows a hexahedral cell, projected onto the image plane. When we refer to a vertex label like F in this figure, we mean either the 3D vertex position or its 2D projection, depending on the context. The projected edges of the cell divide the image plane up into several polygons, in this case the two triangles EIH and BCJ, and five quadrilaterals like $FJCG$. These polygons are scan converted and composited into the image by the graphics hardware. Each of them lies in the projections of a single front-facing face of the cell and of a single back-facing face, so, at least in an orthogonal view, the length l of a viewing ray segment varies linearly across the polygon.

Consider the segment in which the viewing ray through F intersects the cell. The $RGB\tau$ values at the front segment endpoint are the ones determined by the data value at F, but the

values at the back endpoint must be interpolated across the face $HDCG$. The integrated opacity for the vertex F is $\alpha = 1 - e^{-\tau l}$ where l is the length of the ray segment, and τ is the average of the differential opacities τ_f and τ_b at the front and back segment endpoints, respectively. To compute l, we must interpolate the depth z across the polygon $HDCG$ to get the back segment endpoint, and then l can be found using a square root. Similar computations are used at vertex D, with interpolation across polygon $ABFE$ used for the front segment endpoint. The ray segment endpoint values for R, G, B, and τ are interpolated the same way as z.

After a perspective projection, the z_s coordinate in screen space is a function of the coordinate z_e in eye space, of the form $z_s = a + b/z_e$, where a and b are arbitrary, as long as b is not zero, and are usually chosen as functions of the near and far clipping planes, in order to keep the screen space z coordinate within the range of values that can be stored in a hardware z-buffer. This transformation has the important property that planar surfaces in eye space are transformed to planar surfaces in screen space. Thus the depth z_s for the back endpoint of the ray segment through F can be determined by screen space interpolation on the planar polygon $HDCG$, and then l can be found by reversing the eye to screen coordinate transformation. In perspective, l varies nonlinearly with pixel position, so it should be computed by reversing the eye-to-screen coordinate transformation for both the front surface and rear surface ray intersections, and then computing the length l using a square root. This needs to be done at each pixel in polygon $FJCG$, using the z_s values from polygons $FBCG$ and $HDCG$. This requires a square root and two divides per pixel. In our implementation, we approximate this by computing l at each vertex, and interpolating linearly across the polygon. If the projections of the cell faces are small, this introduces only a slight error. We interpolate R, G, B, τ and l linearly across these polygons in screen space, using the standard Gouraud shading and texture coordinate interpolation hardware.

The computations for vertices I and J are simpler, because the necessary R, G, B, τ, and z values are interpolated linearly along the polygon edges. Again, we interpolate these values linearly in screen space, instead of in eye space, but in this case, the correct eye space interpolation is not difficult, because it is only done in one dimension along the front and back edges separately.

In the original Shirley and Tuchman method [46], the color assigned to a "thick" vertex, like F, D, I, or J in Figure 7, is the average of the colors at the front and back endpoints of the viewing ray through F, and the integrated opacity at F is $\alpha = 1 - e^{-\tau l}$ where $\tau = (\tau_f + \tau_b)/2$. The colors at the profile vertices like A come directly from the scalar values s at these vertices, and the integrated opacities are set to zero. The hardware then interpolates the colors and opacities across the image plane polygons like $FJCG$, and composites them onto the image, using

$$RGB_{new} = (1 - \alpha) \times RGB_{old} + \alpha \times RGB_{polygon} \tag{15}$$

Linear interpolation of integrated opacity α across the polygon is not mathematically equivalent to doing the correct calculation, which requires an exponential per pixel. This can cause unwanted Mach bands in the volume rendered image. Correct values of α can be generated

using texture mapping hardware. In an orthogonal view, τ_f and τ_b vary linearly across the polygon in screen space, and therefore so does $\tau = (\tau_f + \tau_b)/2$. If faces $FBCG$ and $HDCG$ are planar, l will also vary linearly across polygon $FJCG$. Thus τ and l can be specified as texture coordinates, and linearly interpolated by the hardware. The correct α is then extracted from a 2D texture map, which is preloaded with the values $1 - e^{-\tau l}$.

For a perspective view, τ_f and τ_b should actually be interpolated in eye space. OpenGL can give a correct perspective of a textured surface, with texture coordinates interpolated in eye space, by using appropriately specified s, τ, and q texture coordinates [66]. However, τ is the average of τ_f on the front face and τ_b on the back face, and the perspective distortion is different on these two faces, so this feature will not help us. In addition, the correct computation of l in perspective would require a square root per pixel. Therefore our texture mapping technique is only an approximation in perspective. We similarly make approximations in the perspective case by interpolating the color components R, G, and B in screen space.

Regarding requirement (d) above, the color integrated along the ray is not the average of the front and back colors, weighted by α as in Equation 15, because the opacity near the front of the ray segment hides more of the back color. For precise color, the analytic integration described in [62, 60] should be performed once per pixel. However, it requires exponentials, square roots, and evaluations of special functions (the Error function or the Dawson integral), and is thus beyond the per-pixel capabilities of current hardware pipelines. Therefore, we instead compute the correct integrated color only at the thick vertices, and divide by the integrated α value to get the vertex colors, which are then interpolated by the hardware.

For the profile vertices, we simply use the color specified by the transfer function. Then we let the Gouraud shading hardware interpolate this color across the polygons, and composite using Equation 15. This gives the correct opacity α at each pixel, and an approximate color that interpolates the effects of the correct integration at the vertices.

We actually have three methods for calculating the chromaticity (color) κ at the vertices. They are, in increasing order of accuracy and computation time: (M1) the average chromaticity $(\kappa_f + \kappa_b)/2$, (M2) the table-based evaluation of (11) described in [60], and (M3) the subroutine-based evaluation of (11) described in Section 5.1 and Appendix A. In methods (M2) and (M3) we divide the radiance from (11) by the opacity α, to get an effective chromaticity at the vertex. When this quantity is interpolated, and used as $RGB_{polygon}$ in the compositing equation (15), the result gives some of the effects one would expect, such as a closer color κ_f partially obscuring a farther color κ_b along the same ray, although it is not as accurate as evaluating (11) at each pixel. We also offer a method (M0), which just uses the hardware bilinear interpolation of α from its values at the vertices of R, instead of the texture mapping. It is even faster and less accurate than method (M1).

Method (M0) is a direct generalization of the method of [46] and may be used if texture mapping hardware is not available. (We have found that the SGI software implementation of texture mapping in OpenGL is inferior in quality to the hardware implementation.) Consider a ray segment s of length $l = t_2 - t_1 = z_b - z_f$, on which τ varies linearly between τ_b at

t_1 and τ_f at t_2. Then the transparency $T = e^{-\int_{t_1}^{t_2} \tau(t)dt}$ on s reduces to $T = e^{-l\tau_{avg}}$. If the chromaticity κ is constant on a ray segment, the integral in (10) reduces to $I(t_2) = (1-T)\kappa = \alpha\kappa$, as explained in [27], so method (M1) is appropriate for cells of constant chromaticity. Method (M2) may be sufficiently accurate for 8-bit-per-color images, if the precomputed tables for the integrals of e^{t^2} and e^{-t^2} are large enough in range and resolution. However, since each integral is looked up individually, the separate exponential factor $e^{(-\gamma+\delta t_2)^2/2\delta}$ may cause exponent overflow or underflow. If this happens (we test for overflow in advance) or if the range of the precomputed tables is exceeded, we revert to method (M1). As explained in Appendix A, method (M3) handles all possible inputs correctly and gracefully.

So far we have discussed the approximations resulting when conditions (a) and (d) are violated, and we now turn to conditions (b) and (c). They are related, since (b) concerns the interpolation of z across the faces, and (c) concerns the interpolation of R, G, B, and τ across the faces as well as inside the volume.

The corresponding interpolation produced by our hardware scheme is hard to determine, because interpolation across polygons is not completely determined in the OpenGL specifications [44]. Instead two possibilities are suggested: (1) divide the polygon into triangles, and interpolate linearly across the triangles, or (2) divide the polygon into trapezoids by horizontal lines through the vertices, and interpolate bilinearly in the trapezoids, or linearly in the case the trapezoid degenerates into a triangle. Our hardware approximation to the integration along the ray segments assumes a further linear interpolation along the ray segments. Thus it is equivalent to piecewise bilinear interpolation in case (1), and piecewise trilinear interpolation in case (2). This piecewise interpolation is view-dependent, since the subdivision into pieces depends on the subdivision of the view plane into polygons, in addition to any further subdivision produced by the hardware. Only in very special viewing situations will it correspond to the interpolation used by the finite element interpolation functions.

Now consider the effect of nonplanar faces. If we subdivide each quadrilateral face by the diagonal from its lowest index vertex, the resulting cell will have more edges and faces, so polygonal subdivision of the image plane by the projections of its edges will be more complex, and take longer to compute. For example, Figure 8 has 25 polygons, instead of the 7 polygons in Figure 7. This will require OpenGL to output more data, and require the hardware to transform more vertices and set up more polygons. However the number of OpenGL fragments (pixels rendered, see [66]), is still the same, because each pixel in the projection still belongs to exactly one polygon. If the cell were instead subdivided into tetrahedra, the fragment count would also increase, because most pixels would lie inside the projections of several tetrahedra.

If only the problem faces are subdivided, the hardware rendering is equivalent to rendering a cell whose faces are piecewise linear or bilinear. This interpolation is determined partly by the hardware interpolation of cases (1) and (2) above, and partly by the depth values interpolated in software at the other endpoint of the ray segment through a thick vertex like F.

In Figure 7, the back endpoint of the ray through F lies in the interior of the face $HDCG$.

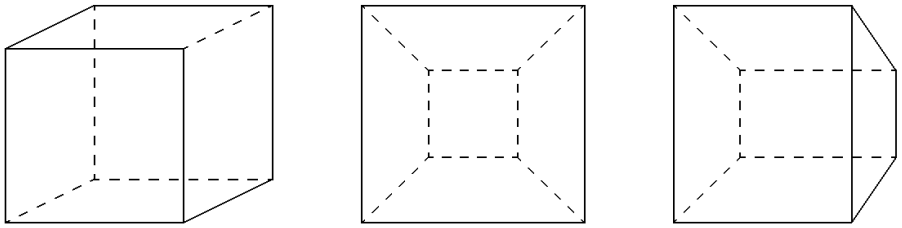

Figure 12: Three topologies from nondegenerate perspective projection of unit cube.

We currently choose the z, color, and τ values at F by subdividing the face $HDCG$ with the diagonal from its vertex of lowest index, and then interpolating linearly in screen space across one of the two resulting triangles. Another possibility would be bilinear interpolation, but doing this correctly would require intersecting the viewing ray through F with the parametrized bilinear surface for the face. This would still not produce the completely correct results for bilinear faces, because the hardware interpolation of the texture parameter l is not the same as doing a ray/surface intersection per pixel.

Now consider the problem of subdividing the projection of a cell into polygons. Shirley and Tuchman [46] used a catalogue of four possible projection topologies for a tetrahedron, and Wilhelms and Van Gelder [55] used a line sweep algorithm for the case of a hexahedron. For a general convex polyhedral cell, we have used an incremental approach to build up a winged-edge data structure [37] for the subdivision. We add the projected edges one at a time, starting with an empty subdivision with a single unbounded face. The new projected edge is extended from its starting vertex, slicing one by one through the existing polygonal regions, and the winged-edge data structure is adjusted accordingly. When all edges have been added, the bounded polygons in the subdivision are the desired homogeneous regions.

We use this hardware compositing of general polyhedra for the slabs of Fig. 3, into which a linear tetrahedron is divided by breakpoints in the piecewise linear transfer functions, as well as for the nontetrahedral zoo elements.

For hexahedral elements, we test whether the element has a projection topology which agrees with one of the three cases shown in Figure 12, which are the nondegenerate perspective projection topologies for a cube. We have precomputed a list of triangle fans for each of these three cases. The vertices for these fans are accessed by the inverse of the permutation needed to map the vertex indices of the element to the standard vertex numberings in the Figure 12 projections. We refer to this approach as *caseification* of hexahedra. This is similar in principle to the work described in [45] but the case classification is more complicated when the cells have nonplanar faces. As well as being more robust, this case analysis is faster than the general cell projection code. However, in the presence of nonplanar faces, other projection topologies are possible, like the one in Figure 13. These occurred quite frequently in the artificially twisted data sets shown in Figures 20 and 36. Currently, if none of the configurations in Figure 12 are recognized for a cell, the general cell projection algorithm is called.

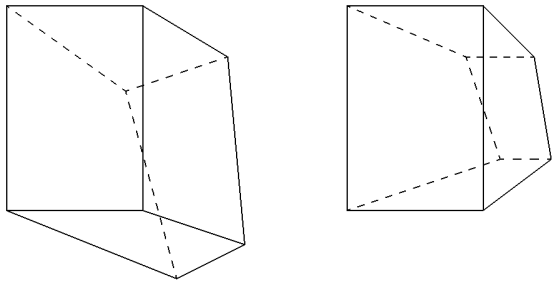

Figure 13: Two common projection topologies for twisted hexahedra.

7 Early Ray Termination

In the HIAC system, in software rendering mode, we use back to front compositing:

For all pixels (i, j)
  $c(i, j) = backgroundColor$
  $\alpha(i, j) = 0$
For all cells Q in back to front order
  For all pixels (i, j) in Q's projection
    compute $c_Q(i, j)$ and $\alpha_Q(i, j)$
    $c(i, j) = c_Q(i, j) + (1 - \alpha_Q(i, j))c(i, j)$
    $\alpha(i, j) = \alpha_Q(i, j) + (1 - \alpha_Q(i, j))\alpha(i, j)$

where $c(i, j)$ and $\alpha(i, j)$ are the values of the color and opacity in the frame buffers.

For efficiency, front to back compositing with early termination could be done on a pixel by pixel basis within cells; this would save unnecessary calls to the numerical routines for the complex error function for those pixels. We have not yet implemented this, but it can be done as follows: invert the sort by reversing the sign of z immediately before the sort and changing it back immediately afterwards; introduce the variable $t(i, j)$ to hold the accumulated transparency at each pixel; then change the accumulation formulas to:

For all pixels (i, j)
  $t(i, j) = 1$
  $c(i, j) = 0$
For all cells Q in front to back order
  For all pixels (i, j) in Q's projection
  If $t(i, j) > threshold$
    compute $c_Q(i, j)$ and $\alpha_Q(i, j)$
    $c(i, j) = c(i, j) + t(i, j)c_Q(i, j)$
    $t(i, j) = t(i, j)(1 - \alpha_Q(i, j))$
For all pixels (i, j)
  $c(i, j) = c(i, j) + t(i, j)backgroundColor$

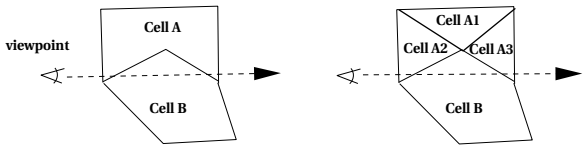

Figure 14: Example of how a problem face between cells A and B cause their visibility ordering to be indeterminate. Note, there would be no problem with the twisted face if the viewpoint were underneath cell B. One way to fix the problem is by subdividing cell A as shown.

$$\alpha(i,j) = 1 - t(i,j)$$

8 The Visibility Ordering Algorithm

The HIAC volume rendering system uses the cell projection method. This method requires a visibility ordering of the cells. We use the exact and efficient *Scanning Exact Meshed Polyhedra Visibility Ordering* (SXMPVO) algorithm described below and in [5], which is an $O(n^2)$ (worst case) method for visibility ordering n arbitrary shaped, nonintersecting convex polyhedra with planar faces, whose visibility ordering does not contain cycles. The faces of adjacent cells need not be aligned, and the meshes may have disconnected portions. The algorithm is an extension of the MPVONC algorithm [58]. A z-buffer is incorporated in the software rendering engine to serve as a witness to the correctness of the visibility ordering.

In the preprocessing phase, the entire data set is read from disk into core memory. The data are stored on disk in the Silo unstructured mesh data format [47]. This format provides information about the cells and the associated vertices in the mesh. Since this only supplies the connectivity between the vertices, HIAC creates other adjacency information as needed. For example, it is necessary to know, for a given cell face, which cells share the face.

In memory, the zoo elements are stored in the structure given in Section 5.4.1, and there is an array `tt[]` of indices of cells, recording the sorted order.

Each element structure has pointers to its vertices and faces. Each face has a pointer to two `Subface` structures, to allow approximation of nonplanar quadrilateral faces as two triangular subfaces and to allow for tetrahedral subdivision of nonsimplices.

Prior to actually sorting the cells, a partial ordering on the cells is created by marking each cell face with an arrow. The arrow indicates which of the two cells sharing the face is in front of the other cell with respect to the viewer position. This is calculated using the plane equation for the face and the position of the viewpoint. Then an initial pass is made through all the faces incrementing the `numbInbound` counter for the cell to which each face's arrow points. Problem cells, such as shown by 2D analogy in Figure 14, are subdivided into tetrahedra if necessary to avoid visibility-ordering cycles, as described in Section 5.4. The cells are topologically sorted from back to front using the partial ordering. (Thus, by the classification in [30], HIAC is a *sort-middle* algorithm.)

HIAC has used a number of sorting algorithms as it has evolved. Formerly, the sorting was done by one of three methods depending on user preference for speed or accuracy. The slowest method was that of Stein et al [50, 62], derived from Newell et al [33, 34]. A faster but still relatively slow method is the BSP-XMPVO algorithm [4]. A simpler and faster method is the MPVONC algorithm [58], which performs efficiently, but an exact ordering is not guaranteed. A new sorting algorithm for the cells of unstructured meshes recently published by Wei et al [53] appears to have a flaw, therefore, we did not consider it for use in this system.

An efficient depth ordering algorithm is described by de Berg, Overmars, and Schwarzkopf [6] which runs in time $O(n^{4/3+\epsilon})$ for any fixed $\epsilon > 0$. However, this algorithm, which is based on a general framework for computing and verifying linear orders extending implicitly defined binary relations, is quite theoretical and is not readily implemented. Further, the SXMPVO algorithm should be faster for most real-world meshes.

Due to the slowness and complexity of implementation of the BSP-XMPVO sort, we developed a new sorting algorithm, the SXMPVO algorithm. This algorithm computes dependencies between each pair of overlapping exterior faces by scan converting them and comparing the depth of each face at each overlapping pixel. While this may appear very slow, it actually is quite efficient. Once the exterior face dependencies are computed, these new adjacencies are then used just as regular adjacencies across shared faces in the regular MPVO algorithm to traverse the cells and produce a final ordering. Recall that a visibility ordering of the cells is an ordering such that if cell $c1$ occludes cell $c2$, then $c2$ precedes $c1$ in the ordering.

Since SXMPVO computes dependencies between exterior faces, it will be convenient for scan conversion if we can assume that all exterior faces are planar. All the cells that HIAC deals with have either triangular or quadrilateral facets. Therefore to assure planar exterior faces, we subdivide all exterior quadrilateral facets into two triangles. Henceforth we call the exterior triangles *subfaces*.

The SXMPVO algorithm works as follows. The b exterior subfaces among the n cells with a total of v vertices in the data set are identified in $O(n+v)$ time and sorted, using quicksort, in decreasing distance from the viewpoint to the centroid of the cell (back to front order), in $O(b \log b)$ time. The sorted exterior subfaces are placed in a queue Q. An array of pointers to `pixelListEntry` structures (see code below) is created, one per pixel in the final image.

```
typedef struct externalSubface {
  Cell *mParentCell;
  char mFaceID, mSubfaceID;
  int verts[3];
  Subface *mSubface;
  float dist;
} externalSubface;

GLOBAL externalSubface *gExternalSubfaces;
GLOBAL int *gCellQueue;
```

```
typedef struct pixelListEntry {
  externalSubface *mExternalSubface;
  double d;
  struct pixelListEntry *next;
} pixelListEntry;
```

The exterior subfaces are then removed from Q one at a time and scan converted. The subfaces are sampled at the same pixel resolution and location (i.e., at pixel centers as per OpenGL [44]) as will appear in the final image. For each projected pixel location *pix-loc* on an exterior subface, a new `pixelListEntry` is created with a pointer to the subface and the distance d from the viewpoint to *pix-loc*. The `pixelListEntry` is then inserted in the linked list of entries for the pixel, in order of decreasing of d.

When all subfaces have been scanned into pixel list entries, the linked list for each pixel *pix* contains a list of exterior faces in depth order, which represent the exterior subfaces which are encountered while traversing a viewing ray from pixel *pix* to the viewpoint. The exterior subfaces referred to by adjacent `pixelListEntry`s in this list therefore alternate between back-facing subfaces and front-facing subfaces.

Our aim is to discover dependencies between exterior subfaces by traversing the pixel lists. Such dependencies are important only across "gaps" in the data set where no cells intervene, because the normal MPVO algorithm already handles ordering between adjacent cells sharing a face. Since each pixel list is ordered from back to front, these gaps occur whenever there are two adjacent `pixelListEntry`s: a front-facing subface followed by a back-facing subface. The first subface in a given pixel list is always a back-facing subface and is discarded. The last subface in the list is also discarded. Then all remaining pairs of `pixelListEntry`s are used to infer information about occluding external subfaces as described next.

For every pixel, the first `pixelListEntry` of each pair refers to a front-facing `Subface` and the second refers to a back-facing `Subface`. The second `Subface` by definition occludes the first and there are no cells between them for this pixel. The second `Subface` is added to the `pseudo-neighbors` list of the first `Subface`, if it does not already exist there. (Note, the pseudo-neighbor relationship between these two subfaces could have already been discovered at an earlier pixel). The `numbInbound` counter for the cell associated with the second `Subface`) is incremented, if the dependency is newly discovered. This count is used by the breadth-first search (BFS) in the topological sort described below.

At this point, each nonexterior facet of every cell has an arrow as described in [58] and in Section 5.4.1, and each `Subface` has a linked list of dependencies on other subfaces discovered by scanning. Each cell's `numbInbound` counter properly indicates the sum of the number of INBOUND arrows and the number of cells which it occludes due to pseudo-neighbor relationships discovered by scanning.

We now can do a BFS of the cells to discover a valid painting order as follows. First, a search is done through the cells to find *source cells* (cells with `numbInbound` $= 0$). These are cells which have no incoming dependencies on other cells. We put all source cells into

a global queue, `gCellQueue`. As explained in [58], cells whose `numbInbound` is zero do not occlude any cells and may be immediately rendered. We then remove the cell at the head of the queue and output it by entering its identifying index into the `tt[]` array, a global array containing the output of the sorting algorithm: the list of cells in visibility order.

We then decrement the `numbInbound` counters of all cells that depend on the cell just output. Any cell which has its `numbInbound` counter decremented to zero is then inserted at the tail of `gCellQueue`. The next cell is removed from the head of the `gCellQueue` and the process is repeated until no more cells remain. If the `gCellQueue` becomes empty and cells remain which have not been output, there must be a visibility cycle in the data and no visibility sort is possible unless one or more of the offending cells is subdivided.

The sort is guaranteed to be correct to the resolution of the final image because of the transitivity of the front-back relationship and the correctness of the individual front-back relations discovered. However, the sort may not be absolutely correct if faces which overlap are not sampled at the points where they do overlap. Since we sampled all faces at all actual pixel locations, the final image is guaranteed to be correct.

This algorithm is nominally still $O(b^2)$, because that is the worst case for the number of new dependencies which exist among exterior faces, and each dependency must certainly be considered at least once. However, the algorithm runs in $O(W \cdot H + A + b + n)$ time in practice, where W and H are the window width and height in pixels and A is the total area of the exterior faces in square pixels. For applications where the area of each exterior subface polygon tends to be fairly homogeneous and where the number of exterior polygons is on the order of the screen resolution or higher, which may be quite common in practice for reasonably large scientific data sets, this is roughly a linear algorithm in b, which one expects to be $O(n^{2/3})$. Insertions into the linked lists for the pixels are almost always at the head of the queue due to the presorting by centroid. Our tests showed that insertions into the pixel queue occur at the head of the queue at least 99% of the time.

Inserting new neighbors into the `Subface`s could potentially be an expensive operation, as the length of each list of neighbors for each face could be on the order of b, the number of exterior faces. Since the entire list must be checked to avoid duplication, this implies a worst-case $O(b^2)$ complexity for the overall algorithm. However, it is likely that, if there is a duplicate dependence already in the list, it will be near the top, having been placed there earlier in the scan conversion of the same face.[3] To further encourage this behavior and to avoid repeatedly finding duplicate dependencies deep in a `Subface`, when a dependency is found to already exist, it is relocated to the top of the list. In this way, insertions will be done in nearly constant amortized time for almost all data sets. Having avoided this potential slowdown, creating the actual dependencies from the pixel lists is $O(W \cdot H \cdot d)$, where d is the average depth complexity of the exterior faces in the scene.

[3]Consider a square face A, which obscures two rectangular faces B and C. Assume B and C share an edge, and the projection of that edge bisects the projection of square A. As we scan a line across A we will discover the dependency of A on B, and then on C, and then move to the next line, whereupon the dependency of A on B is rediscovered, and then on C, etc.

9 Parallelization of HIAC

In this section we first discuss the parallelization of the hardware-assisted cell projection algorithm. Then in Section 9.2 we discuss a parallel algorithm for the high accuracy software rendering mode. Finally, in Section 9.3, we discuss the implementation and performance of a tiling scheme based on a *k-d* tree [51].

9.1 Parallel Hardware-Assisted Cell Projection

Pthreads were chosen over MPI because the Pthreads library uses shared-memory processing, which is natural to the IRIX operating system on which HIAC is currently being developed. Furthermore, process-level parallelism using MPI would dramatically increase message overhead due to its use of network protocols for messaging. MPI would, however, allow for easier porting from SGI to large-scale distributed-memory architectures.

From the perspective of parallelization, HIAC has three phases: (a) sorting, (b) projection and color integration, and (c) executing the actual OpenGL calls. When run in serial with hardware rendering, the projection and color integration phase was the bottleneck. Therefore, we focused our efforts on parallelizing that phase.

During HIAC parallel operation, each cell moves through a pipeline as in Figure 15. One thread is used for sorting since the sorting algorithm does not lend itself to parallelization and because the sort is much faster than the projection phase. Also, only one thread is used for making the OpenGL calls since the graphics hardware is not optimized for multi-thread use. Therefore the cpu horsepower is focused on the bottleneck phase, the projection phase, where multiple threads are used.

The output of the sorting phase (the input to the projection phase) is an array of ordered cells. This array can be decomposed into subarrays and fed to the projection phase as independent work quanta as long as the overall ordering is maintained, which is achieved by thread communication as discussed below.

The sorting thread proceeds to place the cells' indices into an array in back-to-front order as they are discovered by the sorting algorithm. When the sorting thread finishes "enough" cells (this is a tuning parameter) to constitute a work quantum, the projection threads begin to project the sorted cells. The projection threads place their resulting vertex, color, and texture information into another array. The OpenGL thread pulls this information and issues OpenGL calls.

We tested two somewhat different approaches to parallelizing the hardware-assisted cell projection algorithm. Both approaches use threads and the same overall mode of operation as shown in Figure 15. However each approach uses different methods for communication and synchronization as we wanted to see which would give the better performance.

The first approach is called the *tetrahedra-only mode* parallel algorithm because it is limited to projecting and rendering only tetrahedra. It uses the projection and hardware-assisted

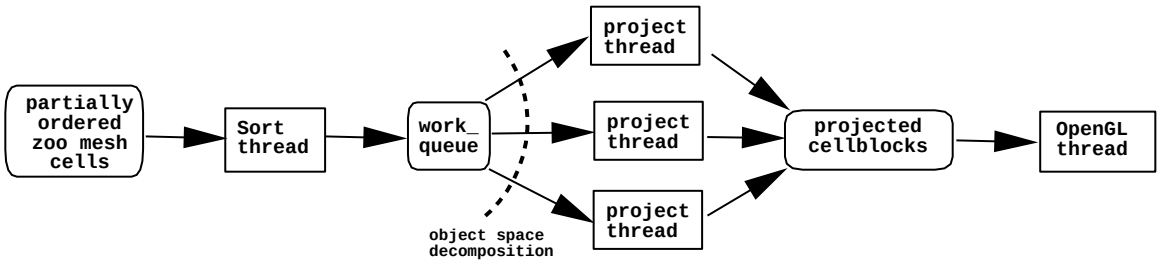

Figure 15: HIAC operation in parallel mode.

rendering algorithms described in Section 6 which are extensions of the Shirley and Tuchman algorithm [46] for tetrahedra.

The second approach, called the *zoo-mesh mode* parallel algorithm, utilizes the projection methods described in Sections 5.3, 5.4, and 6, which subdivides the projected polygons (see Figure 6) into triangle fans.

The next Section discusses discusses the first approach, and Section 9.1.2 covers the second method.

9.1.1 The Tetrahedra-only Mode Parallel Algorithm

When operating in tetrahedra-only mode, two sets of global buffers, whose access is restricted by semaphores, are used for communication between phases. The first of these is composed of two integer arrays, called the *sort_projection* buffers, which are used for transferring cell indices from the sort thread to the projection threads. One of the projection threads, the *communicator*, is selected to communicate with the sort and render threads. The sorting thread sorts a global array of cells in visibility order and stores the sorted indices into one of these buffers. Once this array is full, the *sort_projection* buffers are swapped and the *communicator* wakes up the projection threads, which begin projecting these cells. The projection threads each have a nonoverlapping section of the *sort_projection* buffer that they pull their work from. This is done in order to gain locality of memory references for each thread. The threads project the tetrahedra onto the view plane and decompose their projections into triangles. For each tetrahedron they record all of the vertex, color, and texture information needed to render the polygons. This information is stored in a `splat` record, an element of one of the two *projection_render* arrays.

```
typedef struct {
  int class;
  float texture[MAXPTS*2];
  float   color[MAXPTS*4];
  float  vertex[MAXPTS*3];
} splat;
```

While the projection threads are filling one *projection_render* array, the rendering thread is using the information stored in the other array to make the appropriate OpenGL calls.

9.1.2 The Zoo-mesh Mode Parallel Algorithm

During zoo-mesh operation, communication between sort and projection threads is handled by the sort thread. When the sort thread finishes sorting "enough" cells, it obtains a `workQuantum`, shown next, from the global pool.

```
typedef struct {
  char doProjectFlag;
  long startCell, endCell, pcbIndex;
  long ttStart, ttEnd, pcbIndex;
  char doneProjectingFlag,
inUseFlag, isLastFlag;
  long vvIndex, vvMax, polyIndex,
polyArraySize;
  GLfloat *mTexArray,
*mVertexArray, *mColorArray;
  int *polyVertexCounts;
} workQuantum;
```

A reference to this quantum is placed into two global arrays, which are the work piles for the projection and rendering threads respectively. The sort thread fills in the start and end cell information for the work quantum and then awakens a projection thread to process it. The projection thread performs the projection calculations on the cells referenced by the work quantum and stores the information from its calculations on each cell in arrays within each work quantum. When the work quantum is complete, the projection thread marks the `doneProjectingFlag` for the work quantum and goes to sleep waiting for another work quantum. The render thread continually polls the `doneProjectingFlag` of the next work quantum in its work pile and, when this flag is set, the render thread makes the required OpenGL calls.

We found that it is very important for parallel performance that global variables and mutex calls be kept to an absolute minimum, shared variables be tagged *volatile*, polling strategies be avoided, using signals instead, and calls to *malloc()* be reduced by preallocating a large chunk of memory and distributing it within the program. In addition, it was useful to be able to resize the work quantum size.

9.2 Parallel Algorithm for Software Rendering

HIAC creates its most accurate images by performing a separate exact analytic or numerical integration for each of the three primary colors for each pixel that a cell projects onto. This

can only be done in software. The results of the ray integrations are stored in 3 separate frame buffers, one each for the red, green and blue channels. Therefore, if cell i projects onto p_i pixels, for n cells a total of $\Sigma_{i=1}^{n}(3p_i)$ integrations are needed. For large data sets, this may be very slow. Therefore, we investigated a parallel algorithm on a distributed memory machine, an IBM SP2, using MPI.

In this algorithm, the entire data set is first distributed to all the nodes of the SP2, then for each view point, the nodes run a distributed *k-d* tiling algorithm [51] to create load balanced tiles at each node. No data actually moves; only bounding box statistics are exchanged until a satisfactory balance is achieved. Then each node calculates a visibility ordering and performs volume rendering integrations on its partition of the data, resulting in a tile of the image in the three frame buffers residing on each node. Threads are not used. The green frame buffer tiles are sent to a *green master* node, the red tiles to a *red master,* and the blue tiles to the *blue master* node, via the interconnect switch. The *green master* node then assembles the final green image, etc. Finally, the green, blue and red masters send their images to the overall master node, which then accumulates the three frame buffers into the final RGB image and writes it to disk.

9.3 *K-D*-based Tiling

Tiling can significantly improve the worst case performance of the SXMPVO algorithm, as well as enable HIAC to access multiple graphics pipes and allow the software scan conversion to be parallelized. To accomplish this it is necessary to associate a 3D partition of the data set with each tile such that a partition contains only the cells that project onto its associated tile. Some cells may project onto multiple tiles, in which case they appear in several partitions.

Therefore, we have implemented a tiling scheme based on an optimized load-balanced 2D *k-d* tree [51] decomposition of the data such that each 3D partition contains approximately the same number of vertices. Based on our tests, the use of vertices as a proxy for cells appears to be valid. Therefore the overall result is a set of load balanced partitions associated with each tile containing approximately the same number of cells.

The algorithm starts by sweeping a plane parallel to the *y-z* plane to find the x value where approximately half the vertices lie on either side of the plane. It then repeats this process for a plane parallel to the *x-z* plane, finding a similar y value. The algorithm then chooses the best of these two partitions, i.e the one that gives the best balance, and uses it to start the construction of the *k-d* tree. Alternatively, an optimized *k-d* tree can be built starting with a sweep parallel to the *y-z* plane, and another *k-d* tree built starting with a sweep parallel to the *x-z* plane. Then the tree with the best load balance and redundancy metrics can be chosen.

Timing results for *k-d* tiling are given in Section 10.4.3. It should be noted that this *k-d* tiling algorithm lends itself nicely to a fan-out parallel implementation.

TYPE OF INTEGRATION	NUMBER OF PIXELS	MPVONC SORTING TIME		RENDERING ENGINE TIME		TOTAL TIME	
		240,122 cells	1,403,504 cells	240,122 cells	1,403,504 cells	240,122 cells	1,403,504 cells
Exact Linear	300,000	1.1	9.1	37.2	136.8	38.3	145.9
	600,000	0.8	9.7	50.2	176.7	51.0	186.4
Approximate Method	300,000	0.8	10.4	33.3	124.6	34.2	135.0
With Slicing	600,000	0.9	11.5	44.0	156.4	44.9	168.0
Approximate Method	300,000	0.9	11.2	33.1	125.1	33.9	136.4
Without Slicing	600,000	0.9	12.1	43.9	151.4	44.8	163.5

Table 1: Typical timings for volume rendering **tetrahedra in software** using different integration methods. The MPVONC sorting algorithm is used. All times are in seconds and are from an SGI Power Onyx using one R10000 250 MHz processor. Total time includes viewpoint-dependent setup time.

10 Results

We first present the results for the serial operation of the HIAC volume rendering system: for tetrahedral data sets, for zoo mesh data sets, and finally for zoo mesh data sets with nonplanar faces. Then in Section 10.4 we discuss the results from parallelization and load balancing. Next in Section 10.5 we give timing results for the SXMPVO sorting algorithm. And, finally Section 10.6 presents several color images created by the HIAC volume rendering system.

10.1 Serial Timings for Tetrahedral Data Sets

Timings for the HIAC software rendering engine are given in Table 1 for tetrahedral data sets. Times are shown for the exact linear integration method using the Dawson and Error integrals, as well as for the approximate method described in Section 4. Times are given for images with 300,000 and 600,000 pixels. There is no significant difference in rendering time when several semitransparent illuminated isosurfaces are embedded in the image. Timings for the same data sets but using hardware-assisted rendering as described in Section 6 are shown in Table 2. Times are shown for the M0, M1, M2 and M3 methods. The MPVONC sorting algorithm is used. When creating a volume rendered image of the data set with 240,122 cells covering 300,000 pixels, an average cell projects onto 4 pixels. The average cell of the 1,403,504 cell data set projects onto 3 pixels. For 600,000 pixel images, those numbers are 8 and 6, as expected.

10.2 Serial Timings for Zoo Mesh Data Sets

Table 3 gives typical timings for the HIAC software rendering engine for zoo-mesh data sets, using the SXMPVO sorting algorithm. Timings for the same zoo-mesh data sets but using hardware-assisted rendering as described in Section 6, and *caseification* as discussed at the

METHOD OF PROJECTION	NUMBER OF PIXELS	MPVONC SORTING TIME		RENDERING TIME		TOTAL TIME	
		240,122 cells	1,403,504 cells	240,122 cells	1,403,504 cells	240,122 cells	1,403,504 cells
M3	300,000	1.0	11.8	1.7	14.9	2.8	26.7
	600,000	0.9	10.5	1.7	14.4	2.6	24.9
M2	300,000	1.0	11.3	1.6	13.4	2.6	24.7
	600,000	1.0	8.9	1.6	12.3	2.6	21.2
M1	300,000	0.9	11.8	1.3	11.6	2.2	23.4
	600,000	0.9	12.0	1.3	11.5	2.2	23.5
M0	300,000	0.9	9.5	1.2	9.6	2.2	19.1
	600,000	0.9	11.6	1.2	10.8	2.1	22.5

Table 2: Typical timings for volume rendering **tetrahedra using** the different **graphics hardware** projection methods described in Section 6. The MPVONC sorting algorithm is used. All times are in seconds and are from an SGI Power Onyx using one R10000 250 MHz processor. Total time includes viewpoint-dependent setup time.

TYPE OF INTEGRATION	NUMBER OF PIXELS	SXMPVO SORTING TIME		RENDERING ENGINE TIME		TOTAL TIME	
		750,000 cells	1,500,000 cells	750,000 cells	1,500,000 cells	750,000 cells	1,500,000 cells
Exact Linear	300,000	11.9	28.4	113.5	177.9	128.0	213.3
	600,000	13.3	26.6	176.9	250.8	192.9	283.3
Approximate Method	300,000	12.1	28.3	68.9	129.1	83.7	164.4
With Slicing	600,000	12.4	22.9	87.7	152.9	102.35	180.2
Approximate Method	300,000	11.1	30.3	68.6	130.7	81.9	168.5
Without Slicing	600,000	12.3	26.9	87.6	154.5	102.2	187.3

Table 3: Typical timings for volume rendering **zoo meshes in software** using different integration methods. The SXMPVO sorting algorithm is used. All times are in seconds and are from an SGI Power Onyx using one R10000 250 MHz processor. Total time includes viewpoint-dependent setup time.

end of Section 6, are shown in Table 4. In contrast, Table 5 shows the results of the same executions, but with *caseification* turned off. The latter case is representative of hardware rendering of general cells where caseification may not be applicable. The MPVONC sorting algorithm is use for the hardware rendering timings.

The data sets that were timed consisted of all hexahedral cells with planar facets. No cells were subdivided. When creating a volume rendered image of the data set with 750,000 cells covering 300,000 pixels, an average cell projects onto 17 pixels. The average cell of the 1,500,000 cell data set projects onto 9 pixels. For 600,000 pixel images, those numbers are 34 and 18, as expected.

METHOD OF PROJECTION	NUMBER OF PIXELS	MPVONC SORTING TIME		RENDERING TIME		TOTAL TIME	
		750,000 cells	1,500,000 cells	750,000 cells	1,500,000 cells	750,000 cells	1,500,000 cells
M3	300,000	5.4	16.5	56.5	119.8	61.9	136.2
	600,000	8.0	17.1	59.3	120.7	67.2	137.8
M2	300,000	5.97	16.8	52.7	112.1	58.6	128.9
	600,000	8.2	17.3	55.6	112.7	63.7	130.0
M1	300,000	5.34	15.1	45.1	95.5	50.5	110.6
	600,000	8.2	17.2	47.9	99.0	56.1	116.2
M0	300,000	5.4	15.3	44.5	93.7	49.9	109.0
	600,000	7.4	16.3	46.0	96.7	53.4	113.0

Table 4: Typical timings for volume rendering **zoo meshes using** the different **graphics hardware** projection methods described in Section 6. All cells are hexahedra and **all cells are** *caseified* as described at the end of Section 6. The MPVONC sorting algorithm is used. All times are in seconds and are from an SGI Power Onyx using one R10000 250 MHz processor. Total time includes viewpoint-dependent setup time.

METHOD OF PROJECTION	NUMBER OF PIXELS	MPVONC SORTING TIME		RENDERING TIME		TOTAL TIME	
		750,000 cells	1,500,000 cells	750,000 cells	1,500,000 cells	750,000 cells	1,500,000 cells
M3	300,000	5.4	13.5	411.5	825.9	416.9	839.4
	600,000	6.1	13.3	409.9	829.5	416.0	842.8
M2	300,000	5.9	15.0	392.8	792.1	398.7	807.0
	600,000	5.9	13.2	392.6	790.7	398.5	803.8
M1	300,000	7.4	14.9	375.6	753.1	383.0	768.0
	600,000	7.4	15.7	375.5	758.0	382.9	773.7
M0	300,000	5.4	15.0	377.3	757.3	382.7	772.3
	600,000	6.0	14.9	374.1	757.2	380.1	772.1

Table 5: Typical timings for volume rendering **zoo meshes using** the different **graphics hardware** projection methods described in Section 6. Same data set as previous table, except **no cells are** *caseified*. The MPVONC sorting algorithm is used. All times are in seconds and are from an SGI Power Onyx using one R10000 250 MHz processor. Total time includes viewpoint-dependent setup time.

10.3 Serial Timings for Zoo Data with Cells with Nonplanar Faces

The image shown in Figure 20 was created using the hardware polyhedron projection methods for zoo data sets, using view-dependent subdivision, as discussed in Section 5.4.2. The image shows a volume rendering of a curvilinear grid of 19,000 cells on a half-cylindrical shell, twisted so that its faces are nonplanar. The cells containing contours for the breakpoints of the transfer function are divided into tetrahedra, some of which are further subdivided into slabs on which the transfer functions are linear. In addition, 2589 problem cells were subdivided, giving a total of 31945 cells to be projected and rendered. The subdivision took 0.12 seconds, and the quick approximate MPVONC sort took 0.2 seconds. The total time to project and render the 31945 cells was 14 seconds, using OpenGL and X, and coloring the thick vertices by the simple average of the front and back interpolated colors for their ray segment endpoints. The resolution is 641 by 465. The server was an SGI ONYX with 48 250 MHZ R10000 processors, of which we only used one.

The client was an SGI Octane with one 250 MHZ R10000 processor, and an ESI graphics board with texture option. When run on the client alone, the sorting and projecting time added up to 0.42 seconds, and the total time was 15.4 seconds. The latter increased to 17 seconds when the more accurate color integration was done on the thick vertices, and to 31.6 seconds when all quadrilateral faces were divided up into two triangles for greater accuracy (see Section 5.3). When all cells, whether problem cells or not, were divided up into a total of 114,000 tetrahedra, subdivision and sorting time increased to 1.27 seconds and the total rendering time to 40.7 seconds. Software rendering without dividing nonproblem cells into tetrahedra, but with analytic integration of the correct color on each pixel's ray segments, took a total of 31.2 seconds.

10.4 HIAC Parallelization Results

First we discuss the results of parallelizing the hardware-assisted volume rendering algorithm, and then Section 10.4.2 covers the results of parallelizing the high accuracy software rendering algorithm. Section 10.4.3 gives results from k-d load-balanced tiling.

10.4.1 Results from Parallelization of Hardware Assisted Algorithm

To determine the speedup from parallelizing HIAC, we measured the execution time of HIAC from the moment the sort started to the time the last cell was projected using various numbers of projection threads. We used the MPVONC sorting algorithm for parallel measurements. The results are from execution on an SGI ONYX with 48 250-MHZ R10000 processors.

In order to compare the performance of the tetrahedra-only and zoo-mesh modes, we created artificial data sets of varying sizes, by wrapping a curvilinear grid around a cylinder, as shown in Figure 36. The twisting of the spiral made non planar faces in every hexahedron, exercising the view-dependent subdivision in the zoo element mode.

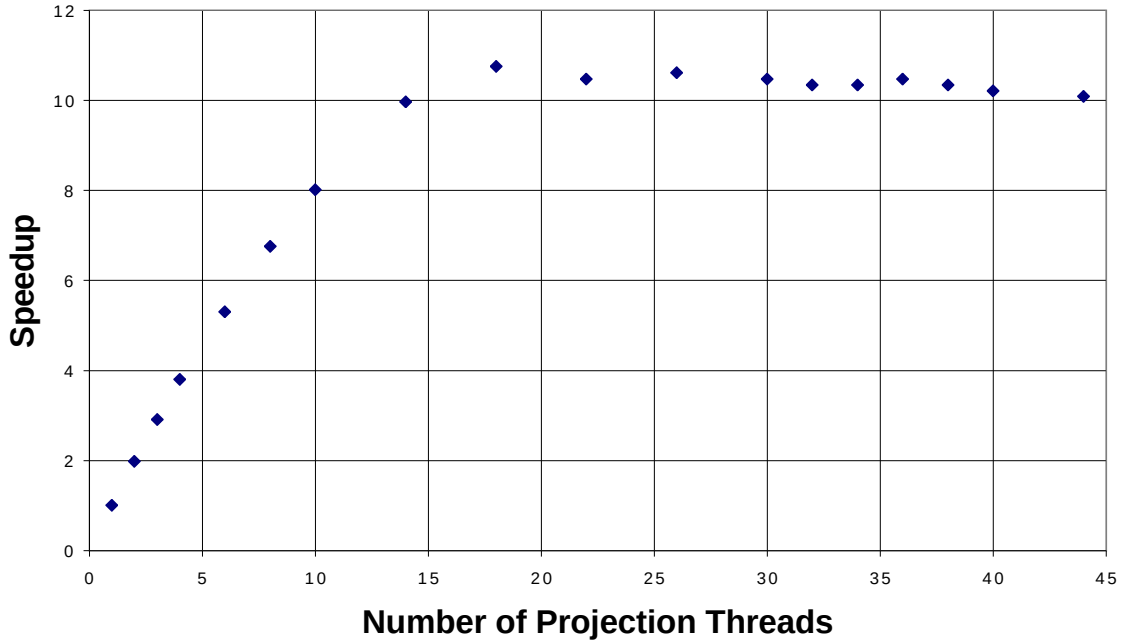

Figure 16: Speedup using zoo-mesh cell projection on a data set of 308,190 cells.

Zoo-mesh Mode: We expected the zoo-mesh mode to generate less triangles and put less burden on the graphics pipeline. For example, the hexahedron projection shown in Figure 6 can be drawn using two triangle fans, with a total of 12 nonoverlapping triangles. On the other hand, when subdivided into six tetrahedra rendered with the tetrahedra-only method, it requires six triangle fans, with a total of 22 triangles. In some places as many as four of these triangles overlap, adding to the OpenGL fragment count, as well as to the vertex and triangle count.

Our tests showed that the zoo-mesh mode had, on average, approximately 2/3 as many triangles as the tetrahedron-only mode. This is not as good as the 12/22 ratio above because the nonplanar faces of some cells had to be subdivided in order to allow a visibility sort. The number of these is view-dependent. The best performance for the zoo-mesh mode was faster than the tetrahedron-only mode, by approximately the ratio of triangles rendered, but this was achieved at the expense of using many more processors, because the general cell projection method was much slower. The relation of rendering time to triangle counts may not be meaningful, however, because neither projection method was able to saturate our graphics hardware pipeline, no matter how many projection processors we used.

When operating in zoo-mesh mode, Figure 16 shows that for up to ten projection threads, speedup (as measured by execution time using one thread divided by time using n threads) is very near linear. Adding more threads results in diminishing returns, until at 18 threads the curve flattens out and there is actually a small slowdown. We believe this decline is due to communication costs related to the NUMA. Table 6 gives overall volume rendering timings in seconds for the use of hardware projection method M0 on a hexahedral data set with 299,000 cells with nonplanar faces using from one to twenty-six projection threads.

	Number of Projection Threads										
	1	2	3	4	6	8	10	12	18	22	26
Time	81.6	41.2	28.1	21.6	15.4	12.1	10.2	9.3	7.6	7.7	7.8

Table 6: Overall volume rendering times using zoo mode parallelism for a data set with 299,000 hexahedral cells, which after view-dependent subdivision had 308,190 cells. Times are shown for the use of various numbers of projection threads using method M0 and *caseification*. In addition to the projection threads, one thread is used for sorting, MPVONC in this case, and one for making the OpenGL calls. All timings are in seconds.

The optimum number of cells to choose per work quantum is a tuning parameter in the zoo-mesh code. It is best to choose neither a very small nor large quantum size. Small quanta are finished very often; therefore, the sorting and projection thread spend a significant portion of their time contending for mutexes in order to establish the workload distribution. For very large quanta, which approach in size the order of the total workload, the projection threads waste time waiting for the sorting thread to finish each work quantum so that they can begin their work, and waiting for other threads to finish when there are no more work quanta left. For the timing given here, there were 2,005 cells per work quantum.

Tetrahedra-only Mode: Figure 17 and Table 7 gives timings in tetrahedron-only mode for four different data sets having from 240,122 cells up to 2,246,250 cells. These include the *F117a*, *fighter*, and *helix* data sets. Images of these data sets are shown in Figures 34, 35, and 36 respectively.

The timings show that increasing the number of projection threads to five or six results in the optimum tetrahedra-only mode runtime, which is roughly half the run time of running the code with only one projection thread. We believe that, when running in parallel with tetrahedra-only mode, the limitation on HIAC performance is the memory bandwidth of the NUMA architecture to send color and vertex information from the projection threads to the rendering thread. Since the zoo code requires more computational effort, this effect is not seen there until more threads are used and more memory bandwidth is generated by the projection threads. This explains why the speedup curve for the tetrahedra-only mode flattens out for fewer processors than for the zoo-mesh mode.

10.4.2 Results of Parallelization of Software Rendering on SP2

The results of the SP2 software experiment are shown in Table 8. The gathering operation is the bottleneck and shows up at around 32 processors. A more sophisticated gathering algorithm should postpone the flattening of the speed-up curve to 64-128 processors. Load balancing time is minimal and requires little communication overhead. A good speed up is obtained by partitioning the sorting and rendering – the superlinear speedup is accounted for by the tiling of the original Stein sort, as described in [62]. Overall, the results were satisfactory and should scale to much larger data sets.

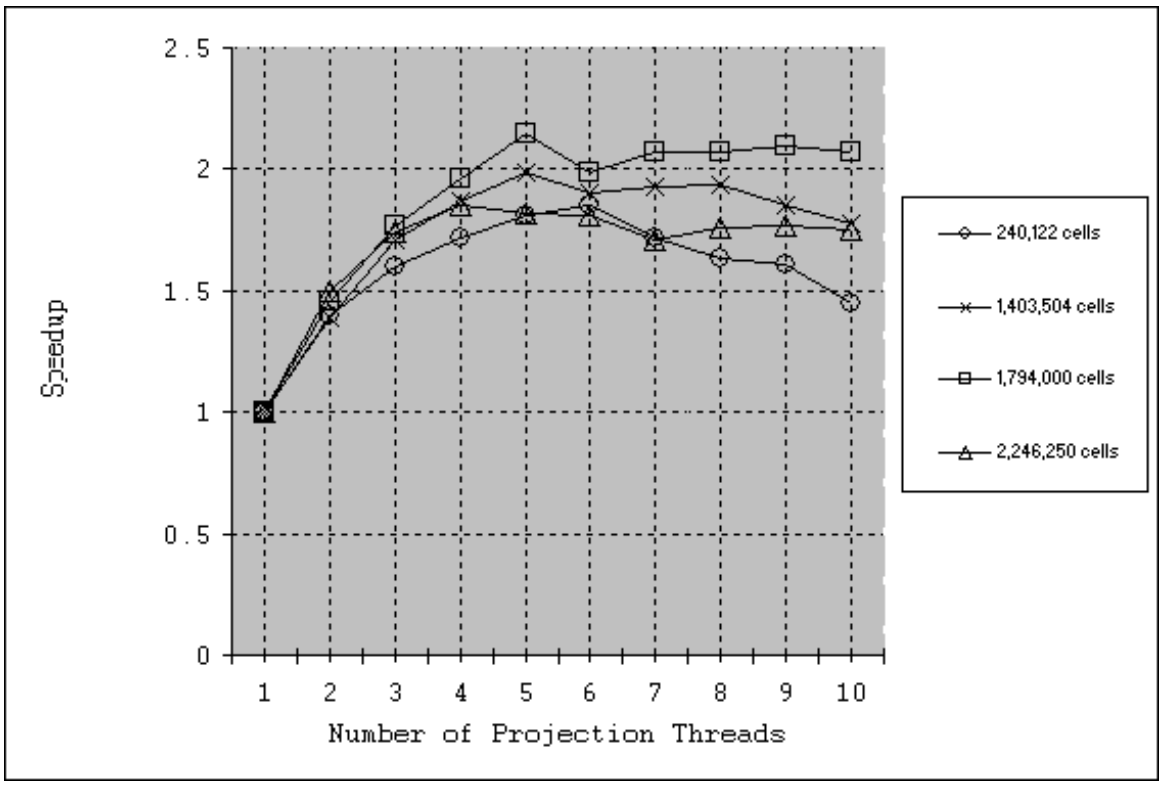

Figure 17: Improvement in performance with increasing parallelism for the tetrahedra-only mode.

number of projection threads	240,122 cells *F117a*	1,403,504 cells *fighter*	1,794,000 cells	2,246,250 cells *helix*
1	3.49	27.66	27.26	32.68
2	2.49	19.79	18.64	21.66
3	2.14	16.14	15.40	18.78
4	2.03	14.80	13.88	17.70
5	1.93	13.93	12.69	17.97
6	1.89	14.59	13.67	18.02
7	2.03	14.36	13.13	19.15
8	2.14	14.26	13.32	18.58
9	2.17	14.94	13.00	18.45
10	2.41	15.54	13.25	18.66

Table 7: Timings for tetrahedra-only mode parallelism. All timings are in seconds.

number of nodes	load balance time	sort & render time	gather time	overall time
4	0.09	39.17	0.43	39.69
8	0.09	13.30	2.40	15.79
16	0.08	6.09	2.75	9.64
32	0.08	2.64	1.52	4.24
64	0.08	3.3	3.09	6.47
128	0.08	1.02	1.05	2.15

Table 8: Results from parallelization of software rendering on SP2. Data set size: 600,000 tetrahedra; all times in minutes.

Number of Cells	Number of Tiles		
	16	32	64
240,000	2.6	5.0	8.7
1,403,504	26.6	50.4	96.5

Table 9: Timing results for *k-d* tiling of unstructured tetrahedral data sets. Times shown are in seconds, using a single R10000 CPU of an SGI Power Onyx.

10.4.3 *K-D* Tiling Load Balancing Results

Timing results for the calculation of load-balance tiles based on a *k-d* tree decomposition of the data set are given in Table 9.

Some statistics for *k-d* partitioning are given in Table 10. The *balance metric* is: $100 - (100 \times (maxCells - minCells)/avgCells)$, where *maxCells* are the maximum number of cells in a partition, and *avgCells* is the total number of cells in all partitions divided by the total number of cells in the unpartitioned data set. (Recall that a boundary cell may appear in more than one partition.) The *redundancy metric* is: $100 \times (totalCellsInAllPartitions - totalCellsInDataset)/totalCellsInDataset$.

NUMBER TILES	240,000				1,403,504			
	Min Cells	Max Cells	Balance	Redundancy	Max Cells	Min Cells	Balance	Redundancy
16	16,014	18,809	84.5%	19.8%	89,576	105,847	83.2%	10.7%
32	8,357	10,776	75.9%	33.8%	45,330	57,439	76.4%	16.8%
64	4,573	6,437	67.9%	54.6%	22,726	31,632	67.5%	25.0%

Table 10: *k-d* statistics showing the minimum and maximum number of cells in a partition, the balance and redundancy metrics are explained in the text.

Algorithm Step	Time (seconds)
Create Partial Ordering	0.26
Subdivide cells	1.95
Build External Face Array	1.29
Sort External Subfaces	0.08
Scan Subfaces and Add Dependencies	0.56
Discover Source Cells	0.17
Do Cell BFS	1.53
Total Time	5.84

Table 11: Breakdown of execution time for the SXMPVO sort for a data set with 372,581 cells and 128,466 vertices.

10.5 SXMPVO Visibility Ordering Results

The time spent in SXMPVO for smaller data sets is in scanning the subfaces and adding new dependencies. However, for medium sized data sets, the actual BFS and creation of the array of external faces starts to dominate the sorting process due to the necessity of searching the cells for the appropriate subfaces. This could be made more efficient by integrating the creation of the external subface array into the program initialization phase and adding more information to each cell to aid in the BFS traversal. The great speed advantage from SXMPVO lies in the relatively small amount of computational geometry involved: calculating slopes during scan conversion of the faces.

SXMPVO proceeds in several distinct phases. In the first, it creates a partial ordering of the cells by marking shared subfaces with arrows, then it subdivides cells as needed to accommodate twisted faces, then it enumerates all external faces and builds an array for them. The heart of the algorithm is when it extends the partial ordering of the cells to include the extra dependencies between external faces. Finally, it must discover the source cells and do a BFS to compute a final total order. Table 11 breaks down the execution time for these SXMPVO phases. Figure 18 shows how each of these phases perform under various workloads ranging from about 100,000 cells to 1,800,000 cells. The time required varies nearly linearly with the workload for these examples.

Other factors than the number of cells impact the sorting time. A careful look at Figure 18 shows an superlinear trend; this is most likely due to the increasing percentage of overlapping cells in the larger data sets (and thus an increase in the average depth complexity), due to the way the data sets were generated. Thus, more dependencies per cell must be created and more external subfaces must be considered per cell. Similarly, data sets with identical numbers of cells but with differing projection areas of external cell subfaces will take different times to scan the faces, even though the number of external dependencies found may be identical. Therefore, it is difficult to exactly depict in a two-dimensional graph the various effects which cell topology exerts on SXMPVO's performance. Given these effects, the near-

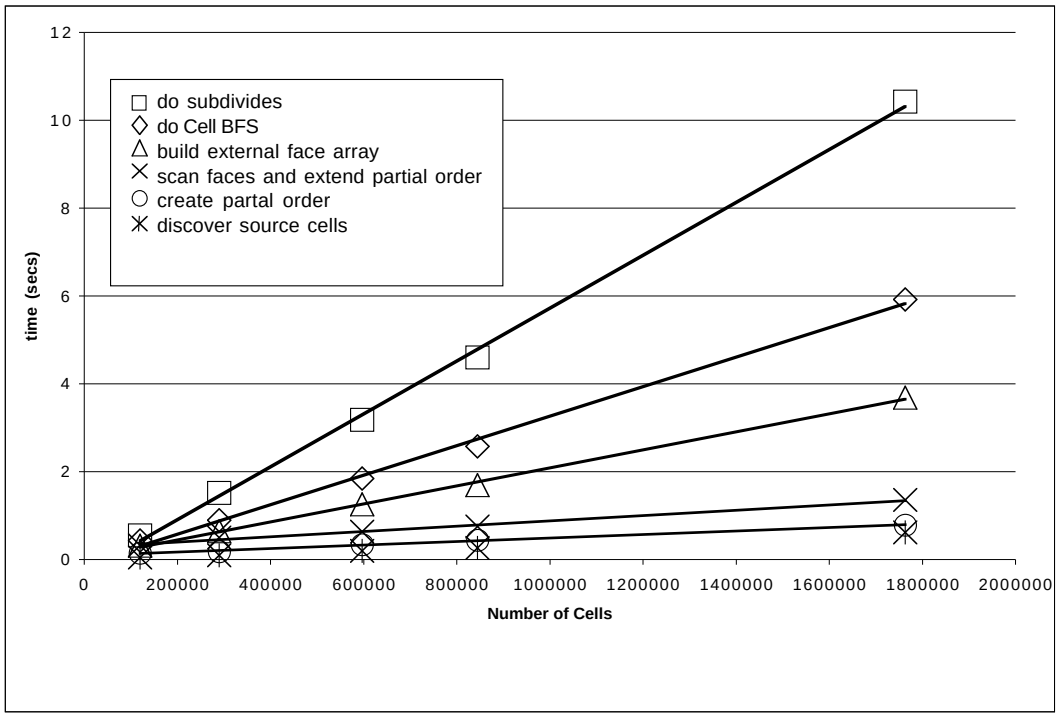

Figure 18: Performance of the different SXMPVO phases with workload.

linearity which Figure 18 shows is remarkable.

The time for both MPVONC and SXMPVO grow approximately linearly in the number of cells. SXMPVO times exhibit a greater deviation (making it appear even sublinear in some cases), due to the algorithm's dependence on other parameters such as depth complexity and image resolution as discussed above. Since SXMPVO must always determine and respect every relationship that MPVONC does as well as creating and following extra relations between external faces, it is always slower than MPVONC for every data set and viewpoint.

10.6 Volume Rendered Images Created by HIAC

As discussed in this report, the HIAC volume rendering system can generate images with several levels of accuracy and speed. From most accurate to least accurate, the methods are:

I. In Software

   A. Quadratic Tetrahedra (exact)

   B. Linear Tetrahedra (exact)

   C. Approximate

   D. Approximate without slicing

II. Using Hardware

A. M3

B. M2

C. M1

D. M0

Methods M0, M1, M2 and M3 may be used with or without slicing. In addition, subpixel splatting may be used with software rendering and the neon and smog optical model may be selected. Many of these options are demonstrated in the images that follow. All of the images shown in this report may be viewed and/or downloaded in color, at their full size and resolution in SGI *RGB* format, for further study at *http://www.llnl.gov/graphics/*.

Figure 20, as discussed earlier in Section 10.3, is a volume rendering of a zoo mesh with nonplanar faces: a half-cylindrical shell twisted so that its faces are nonplanar.

Figures 21 through 30 show volume rendered images of coolant velocity magnitude from a finite element simulation of coolant flow inside a component of the French Super Phoenix nuclear reactor. The data is defined on a mesh of 13,000 quadratic tetrahedra. Figure 21 is an image created using the integration method for quadratic tetrahedra described in Section 5.2. This image is to be compared with the next seven images which were created using the same input specifications as used for Figure 21, but different volume rendering methods.

Figure 22 was generated using the exact integration method for linear tetrahedra described in Section 5.1, by neglecting the data at the interior nodes. Figure 23 was created using the approximate method described in Section 4, which assumes κ is a constant on each ray segment, with the value $(\kappa_f + \kappa_b)/2$; cells were sliced into slabs at breakpoints in the transfer functions as described in Section 5.1. Figure 24 was generated using the approximate method, but without slicing the cells into slabs. Figure 25 was created using the hardware-based polyhedron projection method, (M3), for sliced linear tetrahedra, described in Section 6. Figures 26 to 28 show images created using methods M2, M1 and M0 respectively. Differences between these images are clearly visible in the original images when displayed at their full resolution.

Figures 29 and 31 show volume rendered images with embedded semitransparent illuminated isosurfaces; both were generated using the integration method for quadratic tetrahedra. Figure 30 shows the same view as Figure 29 but was created using the integration method for linear tetrahedra. Figures 32 and 33 show volume rendered images of the density field from a finite element method simulation of air flow past an *F117a* jet aircraft flying at a 20 degree angle of attack. There is a vortex generated, that breaks at the wing trailing edge. This data set is composed of 250,000 linear tetrahedra in a highly adaptively refined mesh. Figure 32 was created using the exact integration method for linear tetrahedra. Subpixel splatting was turned on for the generation of this image; there were 9,200 projected cells covering less than 2 pixels, which were splatted. Figure 33 was created using the approximate integration method, with splatting turned off. Figures 34 through 36 show images created

by the parallel hardware algorithms (tetrahedra-only mode) using method M0. Figure 37 shows an image created using the zoo-mesh code generated in software.

11 HIAC's Current Limits and Future Work

Currently, HIAC renders linear and quadratic tetrahedra accurately using the method described in Section 5. This same accurate technique has not yet been extended to the other zoo elements. This extension can be done as described at the end of Section 5.1, however at present the other zoo elements are dealt with as described in Section 5.3.

SGI multiprocessor machines use a nonuniform memory architecture (NUMA), which means that the time it takes a process to access memory is proportional to the physical distance between the processor and memory. During parallel execution of the HIAC program, data needed by a processor are not necessarily physically close to the processor in memory, but may be several "hops" away. The number of hops a memory reference needs to travel to fulfill a CPU request will tend to increase as the number of processors working on the data set increases. We were unable to find a way to set the SGI API to allow the programmer to specify that a range of memory locations be kept in memory physically near to a certain processor, so this problem is not readily addressable at this time. Thus, there is an inherent limit to speedup on the NUMA due to these extra communications costs.

11.1 Robustness Problems with General Cell Projection

The general cell projection method discussed at the end of Section 6 is not robust, and is prone to numerical problems. When the line segment for a new projected edge is extended from its starting vertex, it may pass through or come very close to one of the existing vertices, or coincide with an existing edge. The algorithm makes numerous tests like "Does point P lie on, to the left of, or to the right of, line L?". A subsequent test, whose answer should depend on this test, may be made again in a slightly different context, for example using the line L' from B to A instead of the line L from A to B. Due to numerical error in finite precision floating point arithmetic, a point P which should lie exactly on line L may not satisfy the corresponding numerical condition, or an inconsistent result may be found for line L'. We try to compensate for this by checking whether the numerical condition is satisfied within an epsilon error tolerance ϵ, set at compile time, but inconsistencies caused by values near these tolerances can still occur.

There are a number of different epsilons, for separate incidence tests like "Does a line pass through a vertex?", "Do two lines coincide (possibly for part of their lengths)?", "Does a new line segment end at an existing vertex?", and "Does a new line segment end along an existing edge?". These tests may return topologically inconsistent decisions about a configuration of vertices and edges, and we have tried unsuccessfully to adjust the various epsilons to give a robust algorithm. One solution to this, to explore in future work, could be to apply exact floating point or integer arithmetic to these incidence tests when the single-word floating

point numbers are within epsilon of the decision criterion. Exact arithmetic should always give consistent decisions. Other alternatives are discussed below.

At several points in the projection algorithm we test for consistency. If an inconsistency is found, we stop generating output polygons, and return with an error flag. Currently this error flag is not tested, so the polygons that should result from the cell may be partially or completely missing. There are other error cases which are not detected by these consistency tests, and can cause a visible error in the image, for example, a bright spot. The Shirley-Tuchman tetrahedron projection code does not suffer as much from these robustness problems, because it classifies the projection into a number of triangle fan cases, using incidence tests which test independent facts about the configuration, and therefore do not generate inconsistencies.

In future work, we should test for the error flag returned by our general cell projection routine. If an error is reported, we should remove the polygons already written to the work quantum (Section 9.1.2) by resetting the address for writing into the cell arrays, and divide the offending cell into tetrahedra. For zoo elements, our existing consistent subdivision code will do this, but for slabs into which a tetrahedron is divided by contour or transfer function breakpoint surfaces, a new tetrahedral subdivision code must be written. Luckily, these slabs are convex, with planar faces, and all the relevant functions of position are linear, so any choice of diagonals for the nontriangular faces (quadrilaterals and pentagons can occur) will give the same correct rendering. Thus a subdivisional algorithm could simple choose one vertex V, subdivide all faces not containing V into one or more triangles, and then take the tetrahedra formed by joining V to each of the resulting triangles.

If the original cell has a degenerate projection, for example, if the projection of a vertex lies on or near the projection of an edge between two other vertices, it is likely that one or more of the tetrahedra into which it is subdivided will have a similar degeneracy. Currently, the parallel version of the zoo-mesh code, discussed in Section 9.1.2, uses the same general cell projection code even for tetrahedra, and will suffer the same robustness problems from degeneracies. The Shirley-Tuchman code for tetrahedron projection is more robust, as discussed above, and should be used for the tetrahedra into which a cell causing an error return is subdivided. This will require going back to the serial version of our enhanced Shirley-Tuchman code, and adapting it to the data structures of Section 9.1.2, because the data structure described in Section 9.1.1 was developed independently, and is inconsistent with the one used for zoo elements.

In addition to suffering from robustness problems, our general cell projection code is slow. However, if not confused by degeneracies, it can project a polyhedron of any combinatorial topological type, not just zoo elements. Thus it could be used for future meshes, like those proposed for the KULL code, which are not restricted to the zoo elements.

For the zoo elements it may be possible to do a more robust case-by-case analysis, as in the Shirley-Tuchman code. As discussed at the end of Section 6, we have made a start on this for hexahedral elements. Future work could include adding more cases, like the one in Figure 13, to the classification for hexahedra, and making similar classifications for the prism and pyramid elements. Cells whose projections were not recognized as one of the

classified cases could be subdivided into tetrahedra, and rendered with the robust Shirley-Tuchman code, once it is adapted to the data structures of Section 9.1.2. Then the general cell projection method, with its robustness problems, would not be needed at all for hardware projection.

11.2 Orientation and Crumpled Cells

The SILO data format does not specify the orientation of the input cells, so each cell must be tested to see whether its faces appear counterclockwise when viewed from outside. This is assumed to be the case when they appear clockwise when viewed from the center of gravity (COG) of the vertices of the cell, which is tested by taking the dot product of the normal to each triangular subface (as computed using the vertex ordering of the triangle in the standard SILO vertex numbering of each the zoo elements) with the vector from the COG to a vertex of the triangle. If all of these dot products are positive, the normals, which will be used in determining the subface plane equations and the directed arrows for the visibility sort, are all correct. If the dot products are all negative, the cell is inside out and all the normals must be reversed.

If some of the dot products are negative and some are not, then the cell is called *crumpled*. In a crumpled cell, there are rays from the COG which intersect more than one subface. If tetrahedra are formed joining the COG to the subfaces, they would overlap, and some would be inside-out. This situation can lead to inconsistencies in both the software and hardware cell projection algorithms, which need a correct classification of front-facing and back-facing faces and subfaces. Currently we just do not draw crumpled cells, and we do not see an easy way to handle them correctly. However, crumpled cells do occur in Lagrangean simulations, so this is a significant limitation of the HIAC system.

11.3 Input File Structure

The hardware and software general polyhedron projection algorithms are not restricted to zoo elements; they can handle arbitrary polyhedra, including those with nonplanar faces, as long as they are viewed in a way that does not introduce visibility cycles or confusing degeneracies. However the current data readers only access data sets of tetrahedra only, or of zoo elements only. In addition, the MPVONC and SXMPVO sorting algorithms work only with triangular faces, so they would either have to be recoded, or, as in our present method for zoo elements, the faces would have to be subdivided into triangles. Neither of these two sorting fixes would be difficult, so the main problem for general cells would be to design appropriate internal data structures, and build the necessary data readers.

Most of the large data sets generated by ASCI simulations are stored in multiple files, as produced by the multiple nodes in the parallel simulation. In the multiple files, there is no global indexing of the vertices. The current data reader in HIAC only deals with single SILO files and then only with UCD-based meshes, and assumes that each vertex has a

unique number. Further, HIAC currently is hard-coded to look at only the first scalar field in the SILO file. Mark Duchaineau has written a general utility called MPACK to read these multiple data files, match the vertices to determine a global vertex array with each unique position represented once, and match the faces sharing the same vertices into a global face array with each face also represented only once. We will need to either need to modify the output from this system to make one single huge file, or else, preferably, interface our data structures to the MPACK reader. For very large data sets, we may not be able to fit the whole data set into memory, and may have to write an out of core alternative for our sorting and compositing steps, which would be quite difficult. Perhaps instead we could distribute the data to multiple nodes, each of which was responsible for a cubical or rectangular solid subset of the data space, as described next.

11.4 Distributed Processing of Partitioned Data

For a distributed implementation where each node has one or more partitions of the data, and not the entire data set, it is not possible to generate a single compositable image for each partition unless every viewing ray intersects the partition in a single connected segment, as would be the case if each volume partition was convex.

Our initial plan was to deal with each partition, hoping it was reasonably compact so as to limit internode communication during the simulation. Then, for each pixel, we can generate locally a sorted list of connected viewing ray segments. Given a back to front sort of the cells in the partition, as produced by the SXMPVO algorithm, our current software scan conversion can detect the gaps between segments of the viewing ray, caused by the ray entering another partition or leaving the data volume entirely, and produce these ray segment lists without further sorting. Alternatively, without any initial sort, or only an approximate one, we could insert each segment in the proper place in its pixel's list, merging and compositing adjacent ones if appropriate, or else do a per-pixel sort and merge/composite when all the cells in the partition have been processed. Finally, these sorted lists would be sent to the node(s) creating the final color image for each pixel. The gathering nodes would then have to merge all the sorted lists of ray segments for each of its pixels before doing the compositing of Section 7. Our code for this, which we call *pixel-chunking*, is not yet complete.

An alternate plan is instead to divide the data volume, perhaps adaptively and/or hierarchically, into rectangular solid regions in screen coordinates, and distribute the cells to the nodes whose volume regions they overlapped. Each such region would project to a single rectangular screen window, and any cells whose projections extended outside that region could easily be clipped along the screen x and y directions to the screen window, using software or hardware. If the volume regions did not overlap in depth, so that each pixel belonged to a single tile, this clipping would be sufficient. However, if z clipping is required, only our software projection algorithm is currently capable of doing this. The standard z clipping in hardware cannot correctly clip the polygons generated by the cell projection scheme, because the ray segment whose integral is reproduced approximately by the hardware has both a front and a back endpoint, and only one of these can be represented in the

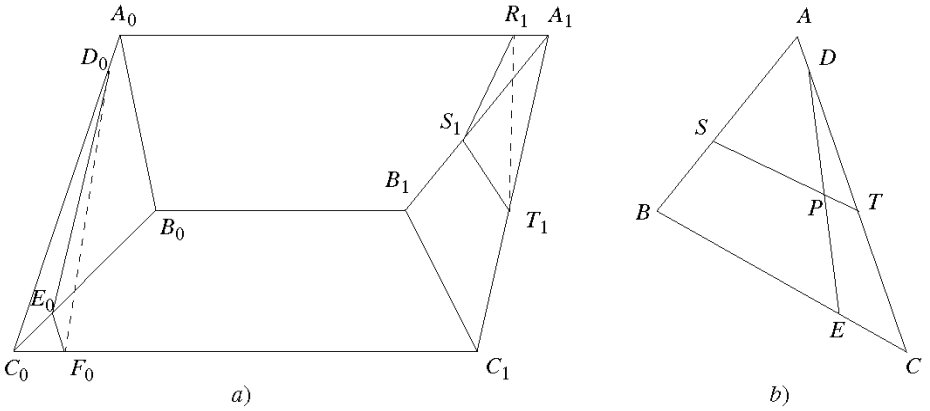

Figure 19: Figure a) is a view of a prism in a direction approximately parallel to the image plane. Figure b) shows an orthogonal projection of the prism onto the image plane.

z value. Instead, each cell would have to be z-clipped in software, before being sent to the cell projection algorithm. Such clipping is not difficult in principle. Each face is clipped in its 2D plane, and then two capping faces are added bounded by the newly created edges in the front and back clipping planes, respectively. However, such clipping could greatly increase the number of topologically distinct cell types and cell projections occurring, which would defeat our goal of replacing the current general cell projection algorithm with a more robust case-by-case use of preanalyzed triangle fans. If such z-clipping is needed, it may be necessary to subdivide all cells into tetrahedra before clipping is done. This will help because then there will be fewer projection cases to deal with.

There is an alternative reasonably efficient way to do the z-clipping in software, if the x and y clipping are done in the hardware. Use the triangle-fan methods above to divide all screen projection polygons into triangles. Then the unclipped 3D volume represented by each triangle is an infinite triangular prism, for the orthogonal case, or an infinite triangular pyramid, in the perspective case, made finite by slicing against both the near and far polyhedron faces projecting onto the triangle. Figure 19 a) shows one such prism for the orthogonal case, viewed in a direction approximately parallel to the image plane. Figure 19b) shows an orthogonal projection of the prism onto the image plane. The prism has three quadrilateral faces orthogonal to the image plane, and two triangular faces, $A_0B_0C_0$ and $A_1B_1C_1$, arising from front and back surfaces, respectively, of the original cell. The front clipping plane slices the prism at triangle $D_0E_0F_0$, and the back clipping plane, at triangle $R_1S_1T_1$. On the image plane, the projected edges DE and ST slice the triangle ABC into three triangles and one quadrilateral, all meeting at P, which can be processed as a fan of seven triangles around P. Inside each of these subtriangles, the corresponding subvolume of the clipped prism is bounded by a single front and back plane, either a clipping plane or the plane of one of the cell's faces, and can thus be handled by our hardware rendering scheme. So the original triangle fan decomposition will be terminated whenever a clipped situation is encountered, and the clipped triangle in question will be replaced by a the new triangle fan.

OpenGL allows the specification of 6 additional clipping planes *glClipPlane()* in hardware

that do not need to be view-aligned. Wittenbrink [64] describes how these planes can be used to correctly project tetrahedra that may overlap more than one partition in the z direction.

11.5 Image Space Tiling

Tiling will allow the use of more than one graphics pipe, giving HIAC extra parallelism and the ability to drive a projection wall.

As described in Section 9.3, HIAC has code to do tiling based on a load-balanced k-d tree [51] decomposition of image space, but the remainder of the system has yet to take advantage of this tiling. Implementation details of the k-d method are given in Section 9.3 and timing results are given in Section 10.4.3.

11.6 Other limits and extensions required

11.6.1 Memory

We have observed that on our SGI Power Onyx which has a total of 15.9GB of main memory, the largest zoo data set that HIAC will execute has approximately 2,500,000 elements. It is not clear exactly how much of that 16GB of main memory that the operating system allows one process to use.

The zoo code and the new SXMPVO sort makes heavy use of *malloc()* during execution. This may account for the wide range of timings for executing identical data during serial mode. This is due to the fact that a call to *malloc()* effectively forces system-wide mutual exclusion, therefore if the system is in heavy use, even in serial mode, the zoo code can become less efficient. This effect may also have a significant influence on parallel performance of the zoo code.

11.6.2 Pointers

The data structures described in Section 5.4.1 were designed to be very compact in order to save as much memory as possible. However, this causes a significant reduction in efficiency due to having to follow many pointers to get the information needed to project and render a given cell.

There is another problem related to the pointers some of which are used to access the vertex coordinates, and to express the topological connectness of the mesh. For historical reasons, the code is a mixture of Fortran and C, and in order to pass pointers through the Fortran calling sequence, they have been converted to 32 bit integers. The current code is thus limited to the 32 bit addressing of the SGI *n32* compiler option. To use 64 bit addressing, with the *n64* compiler option, addresses will have to be cast to long integers instead, so the code will need to be modified to take advantage of the full 16 Gbyte memory of riptide.

Another possibility is to convert all the Fortran code to C or C++, which will be very time consuming as there is a lot of mathematically complex Fortran code. If an automatic Fortran to C translator is used, then the C code may not be very readable and maintainable.

Probably the ideal solution would be to completely redesign the HIAC using the Unified Modeling Language (UML) and re-implement it using an object-oriented language. However, due to the complexity of the mathematics, the programmer would have to have a good understanding of applied math.

11.6.3 Changes in Toplology, Geometry or Viewing Parameters

We would like our system handle Lagrangean meshes, which can deform (change geometry) during a simulation, and even be remeshed (change topology) when necessary to preserve numerical accuracy. In addition, there are other things which can change when interactively viewing a sequence of time steps in a simulation. Listing in order of decreasing impact on the sorting and rendering computation, they are:

a. The mesh topology.

b. The mesh geometry.

c. The viewpoint.

d. The scalar data to be visualized.

e. The transfer functions.

Our file format for storing zoo meshes does not include the topological information necessary to build the adjacency graph, so if the mesh topology changes, we must search for the adjacent faces and build the graph. If the geometry changes, we must recompute the face plane equations. If the viewpoint moves, we must redetermine the directions of the edges in the adjacency graph, (i.e. the arrows) and determine which cells are problem cells. The problem cells are subdivided into tetrahedra, and those that are no longer problems are restored to their original state. For the SXMPVO sort, we must also redo the scan conversion of the exterior faces. If only the scalar data or the transfer function changes, the sorted list can be reused in the rendering. Currently the system does not reuse any information from previous frames; each frame starts from scratch. Therefore, it would greatly improve the efficiency of HIAC, especially for use with time-varying data or Lagrangean meshes, if the algorithms and data structures were modified to reuse as much information as possible.

11.6.4 Graphical User Interface

The system is not intended to be highly interactive, but rather to operate in batch mode to create high quality/accuracy images for careful in-depth study that can be trusted, for publication or for animations. Appendix B shows an example HIAC batch input file which

shows some of the available commands. It would be nice if a graphical user interface were developed (or if HIAC were incorporated into the Terascale Browser) to facilitate the selection of the input parameters and the creation of the image specification file, but not to replace the batch file since the batch input file serves as an excellent archive of the specifications of how the image was generated. In fact, the input files for the images shown in this report are given in Appendix B and can be used by other researchers to reproduce similar images for comparison.

11.6.5 Early Ray Termination

It may be possible to increase the efficiency of the software version somewhat by the use of front-to-back compositing with early termination, as described in Section 7. This can be done on a pixel by pixel basis within cells, and will save unnecessary calls to the numerical routines for the complex error function for those pixels.

11.6.6 Analytic Antialiasing

As mentioned at the end of Section 5.5, our current subpixel splatting method is not always correct, and we are also working on analytic antialiasing, using an exact geometric subdivision of the image plane by the projected edges of all cells. This is done by subdividing the image plane by the projections of the edges of all the cells, into regions where all viewing rays cross the same mesh faces and therefore have expressions for the color integration which vary analytically over the polygon, and can be convolved with a presampling filter. This subdivision is done incrementally, adding edges one at a time to a winged-edge data structure. It unfortunately suffers from the robustness problems discussed in Section 11.1, which could perhaps be solved by exact arithmetic.

11.6.7 Other Algorithmic Optimizations

At present, Williams' [58] MPVO visibility ordering algorithm, which is a heuristic for non-convex meshes, may be used for the hardware assisted previewer described herein. However, the MPVO algorithm has a large storage requirement for its preprocessed data structures, therefore it would be useful to investigate replacing this algorithm with a different sorting heuristic for the cells of unstructured meshes, such as one that sorts the cells by their centers of gravity. For data defined on 3D Delaunay triangulations, Karasick et al. [18] and Cignoni, et al. [2] describe an efficient exact sorting algorithms based on sorting the tetrahedral cells by their powers. This same idea could be extended to non-Delaunay meshes. Wittenbrink [65] describes such a sorting algorithm and shows it to be significantly faster than the MPVONC algorithm.

In addition, Wittenbrink describes a number of cell projection optimization techniques that could significantly improve volume rendering performance for tetrahedral data sets or for data sets that are converted into tetrahedra.

Most of these optimizations come at the expense of image accuracy, therefore it would be useful to compare images generated using Wittenbrink's techniques with images generated by HIAC to see exactly what information is being lost. Even if the faster sorting algorithm turned out to be significantly less accurate, it might be useful as a fast previewer and for setting transfer functions, etc.

In another paper [64], Wittenbrink discusses parallel volume rendering algorithms for unstructured data on PixelFlow [31] which generalize to other compositing architectures.

Another very promising method which should improve the speed of approximate volume rendering of unstructured data is described by Farias and Silva in [9]. This method, a parallelization of the ZSWEEP algorithm [7] for distributed-shared memory machines, which divides the image into tiles, is cache friendly, has fast rendering times, and yields good speedups. The ZSWEEP algorithm sweeps the data with a plane parallel to the screen, projecting the faces of cells incident to the vertices as they are encountered by the sweep plane. This method is of course less accurate than Wittenbrink's method described above, but faster and parallelizable.

Another fast method for volume rendering unstructured data, based on accumulating projected faces, is described by Lucas [22] and is the volume rendering method used by IBM's *Data Explorer*

Out-of-core volume rendering of unstructured grids is described by Farias and Silva in [8]. It would be very useful to investigate the incorporation of this into HIAC.

11.6.8 Embedding of Solid Objects

It is a standard trick to combine transparent objects with precomputed color/z buffers of opaque geometry, by using the z test to decide which OpenGL fragments to composite, but not revising the z buffer value if the test is passed. (Actually, if the transparent objects are visibility sorted from back to front, it does not matter is the z buffer value is rewritten.) This method requires that the semi-transparent polygons be supplied with a z value, which is currently not done in the zoo element version of our code. The z chosen could be the average of the front and back endpoints of the ray, which varies linearly across a subdivision polygon of the projection, since each such polygon lies in the projection of a single front facing cell face (or subface) and a single back facing one. Therefore the linear interpolation of the vertex z values by the hardware is appropriate. However, as discussed above in Section 11.4, the actual z for a ray segment is a z range, and in general the range is truncated but not wholly obscured by an opaque object at a certain z buffer depth, so the all or nothing decision of the z test is not strictly correct. In hardware, the most we can hope for is an approximation which is fairly good if the individual cells are small (at least in z extent) and not very bright or opaque. Correct clipping can only be done in software scan conversion, where the ray segment for each pixel can be correctly truncated, or by clipping each cell in software against the opaque geometry, before doing the cell projection and hardware scan conversion and compositing of Section 6.

11.6.9 Extending High Accuracy Rendering to Other Cell Types

The HIAC volume rendering system described in this report creates highly accurate images of unstructured data sets whose cells are either linear or quadratic tetrahedra. The system was specifically designed to deal with data sets from the finite element method, but it is not limited to this type of data. Currently, the HIAC visibility ordering algorithm and the rendering engine will handle tetrahedra, bricks, prisms and pyramids, or any combination thereof (zoo meshes); but will only use high accuracy integration for linear and quadratic tetrahedra. In addition, HIAC will handle zoo elements with nonplanar facets.

It would be good to extend the high accuracy methods to the other zoo elements and to quadratic brick elements using the approach outlined in Section 5.2.2.

11.6.10 Porting to LINUX

Most of the Fortran code in the HIAC system has been successfully translated into C with *ftoc* an open software Fortan to C translator. It would be nice to complete this port so that HIAC could run under LINUX.

12 Ways to Extend and Adapt the HIAC Volume Rendering System for Future ASCI/VIEWS Architectures

We start with a statement of our understanding of what the general architectural features of the more powerful future ASCI/VIEWS platforms might be. Then in Section 12.2 we discuss how the HIAC Volume Rendering System might be adapted or extended so as to be deployed on such an architecture.

12.1 General Features of Future ASCI/VIEWS architectures

High-end future ASCI/VIEWS computing platforms are expected to include distributed-memory multiprocessors with several thousand nodes, where each node is a distributed-shared memory multiprocessor with from about two or three to a few dozen processors. Some or all of the nodes may or may not have graphics hardware.

The current parallel implementation of the HIAC volume rendering system is optimized for a single SGI Onyx, a distributed-shared memory multiprocessor machine, with 48 processors, of which we generally use at most a dozen or so. This SGI has four Infinite Reality rendering engines of which we use only one. From our perspective, the projected ASCI/VIEWS architecture looks like several thousand machines, each very similar to an SGI Onyx, connected by a fast interconnect network. However, each node may not have graphics hardware. In

addition, each node may have differing amounts of RAM and may have differing numbers of processors.

So the question is "Can the HIAC volume rendering system scale up?" That is, "If HIAC currently take 8-30 seconds to generate an image of a data set with 2,000,000 cells on a single SGI Onyx, then in a distributed environment with 1,000 nodes similar to the Onyx, can an image of $2,000,000 \times 1,000 = 2,000,000,000$ cells be generated in approximately the same time, i.e. 8-30 seconds?" In addition, we want HIAC to be tunable in terms of dealing with nodes with varying numbers of processors and amounts of memory, and where some or all nodes may not have graphics hardware.

If there are n nodes, larger data sets than approximately $2,500,000 \times n$ cells can be volume rendered by either increasing the number of nodes, increasing the storage and time efficiency of HIAC, or assigning more blocks to each node and allowing for a longer rendering time.

12.2 How HIAC Could be Extended/Adapted to be Deployed on Future ASCI/VIEWS Architectures

A number of extensions to the HIAC volume rendering system that would be very valuable in the context of future ASCI/VIEWS architecture were discussed in Section 11, especially in Sections 11.4 and 11.6.7. In this section, we address the most significant modifications to HIAC to permit distributed volume rendering.

The issues we address in this section relate to "Will the HIAC volume rendering system scale?" and "Is it tunable"". We now discuss the following areas:

A. Preprocessing and distribution of data.

B. Internode communication and subimage gathering.

C. Tunable parameters:

   i. Nodes with varying numbers of processors and amounts of RAM.

   ii. Possibility of no graphics hardware.

(A.) **Preprocessing and distribution of data:** One way to perform distributed volume rendering (alluded to in Section 11.4) when the entire data set is not available at each node, is to subdivide the data set into rectilinear paralellopipeds or *blocks* (MPACK files) as a preprocessing step. Using a three-dimensional *k-d* decomposition, the blocks can be defined so as to contain approximately the same number of cells thereby making load balancing possible. The blocks can then be distributed randomly over the nodes, also in a preprocessing step, so each node has approximately the same number of blocks.

By making the number of blocks larger than the maximum number of nodes, the load could be evenly distributed over the nodes regardless of the number of nodes actually being used at any one time, and would allow for nodes that may have more or less RAM memory or

processors than other nodes (by giving the weaker nodes fewer blocks). In addition, by designing each node to accept multiple blocks, which the node could process sequentially, the size of the maximum data set that could be processed would be virtually unlimited. Of course increasing the data set size would at some point fail to scale, but at least it would work, albeit more slowly.

Then at run-time, each node would generate a volume rendered image of the data in its block(s), a *subimage*(s) of the image of the entire data set. Gathering and accumulation of these subimages could be done efficiently over the interconnect network. It is straightforward to visibility order the blocks in a *k-d* tree using the algorithm given below which starts by calling `SortNode` on the root node of the *k-d* tree; *N.plane equation* is the plane equation of the cutting plane used at node N. Note that if the viewpoint lies in the plane used to define the node, then neither subtree block occludes the other, and it doesn't matter which goes first; the second `else` clause then just chooses the right subtree.

```
SortNode(N) {
  if N is a leaf, output N
  else P = N.planeEquation;
     if viewpoint is to right of P
        sortNode(N.leftChild);
        sortNode(N.rightChild);
     else
        sortNode(N.rightChild);
        sortNode(N.leftChild);
```

It should be noted that the z clipping required by this partitioning approach will need the special attention discussed in Section 11.4. Further, for terabyte data sets it will probably be necessary to use an out-of-core algorithm to create a 3D *k-d* decomposition.

It will be useful for the *k-d* tree to have a hierarchical structure so blocks of differing sizes could be distributed to the nodes. For example the first level of a 3D *k-d* tree will have 8 blocks, the second level will result in 64 partitions, the third in 512 blocks, so in general, a level n *k-d* tree will have 2^3n blocks at its lowest level. Therefore, we want the preprocessing step to generate the deepest possible *k-d* tree so the minimum block size is suitable for the node(s) with the least memory. Nodes with more memory could then receive larger blocks from higher levels in the *k-d* tree.

During the preprocessing step of subdividing the data, it will be valuable to tag each block with extracts: information that would be useful in locating relevant blocks in the case of zooming, etc., such as bounding box information.

(B.) **Internode communication and subimage gathering:** Under our proposed scheme, each node could be expected to generate a number of subimages, all part of the overall image, but not necessarily contiguous. These subimages could be sent over the interconnect network for accumulation as they are generated, or as a group when all are completed. The latter approach would result in reduced congestion on the interconnect. An algorithm will need

to designed to efficiently gather the subimages, probably using a fan-in scenario, and to accumulate them using alpha compositing. No other internode communication would be needed except to broadcast the viewing parameters.

(C i.) **Tunable parameters: Nodes with varying numbers of processors and amounts of RAM:** The issue of dealing with nodes with varying numbers of processors and amounts of RAM was addressed in (a) above. Currently, HIAC has the capability to run with tunable parameters for the number of threads and processors.

Related to this is the need for careful thought about how to make the main data structures for HIAC more efficient. Both in terms of their size and in terms of being able to access the required data to project a cell with a minimum of page faults, cache misses and pointers followed. This a key problem that could significantly improve the overall efficiency of HIAC.

(C ii.) **Tunable parameters: Possibility of no graphics hardware:** Given enough processors, the work currently done in graphics hardware could be done in parallel in software without significant loss of efficiency. In addition, this procedure would eliminate the need for read-backs from the frame-buffer in hardware, which can be a slow operation on some graphics cards, but which is necessary in order to gather and accumulate the subimages. Currently, HIAC can run entirely in software but its performance may not be as good as when graphics-hardware is utilized.

12.3 Will HIAC Scale?

It appears that the HIAC volume rendering system will not scale much larger on a single machine, limiting data set size to approximately 3,000,000 cells. However, if larger data sets are partitioned into blocks, as described above, and a distributed system is used, HIAC should scale very nicely. For n nodes, the maximum data set size should be on the order of $n \times 3,000,000$ cells, attenuated by a factor of from 10% to 20% due to duplication of cells (cells that overlap partition boundaries). The sorting time should remain roughly constant as only the cells in a partition need to be sorted — a global sort is not needed. In fact, this should apply to all times. In other words, assuming the data is partitioned and distributed over the nodes in an intelligent manner in a preprocessing step, the time to render $n \times c$ cells on a distributed machine with n nodes should be on the same order as the times given in this report to render c cells. However, this does not include the time to gather the subimages from the nodes and accumulate them into one image — that time is not known. To maintain high accuracy images, the problem of dealing with cells that overlap more than one partition needs to be dealt with; several approaches to this problem are described in Section 11.4.

13 Conclusion

It should be noted that the HIAC volume rendering system was designed to generate reliable and trustworthy high accuracy images. It does this very successfully. It was never intended

as a visual browser or as an interactive technique. In the context of ASCI terascale scientific data analysis and visualization, the HIAC volume rendering system is really a full-service tool that may be applied to a particular space-time neighborhood of interest, rather than to the entire data set. It is quite possible that the full value of HIAC will not be apparent until some time in the future when the ASCI program reaches maturity and the focus is on understanding the physics by careful analysis of the data sets, rather than at the present time where the focus is on the design and development of simulation software and tools.

Acknowledgements

We are grateful to Roger Crawfis of Ohio State University for his contribution to the *SCAN-VOL* code which was modified for use in the system described herein. We appreciate technical assistance from Barry Becker of Silicon Graphics Inc., Cliff Stein of STMicroelectronics, Kwan-Liu Ma at ICASE, and Mark Duchaineau and Randall Frank at LLNL. Janine Bennett developed, implemented, and tuned the tetrahedron-only parallelization code; Debbie May developed an initial implementation of this code. Rich Cook designed, developed, and implemented the zoo-mode parallelization, as well as the new SXMPVO sorting algorithm. The first author is greatly indebted to Sam Uselton and Tom Lasinski at NAS, NASA Ames Research Center for their generous summer support for three years and for equipment loans which supported his early work on this project. He is also grateful for the use of the Large-Scale Interactive Visualization Environment (LIVE) at NAS, NASA Ames which was used to generate some of the images in this report. Both authors received summer support, arranged by Becky Springmeyer and Randall Frank, from the Accelerated Strategic Computing Initiative (ASCI). Robert Haimes of MIT graciously provided the *F117a* data set and Bruno Nitrosso at Electricité de France provided the Super Phoenix data set. The *fighter* data set was provided by David Marcum, ERC, Mississippi State University. This work was partially performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract number W-7405-ENG-48.

Appendix A

This appendix gives implementation details for the evaluation of (11) of Section 5.1. In the notation of that section, let $f_1 = \frac{\gamma+\delta t_1}{\sqrt{2|\delta|}}$ and $f_2 = \frac{\gamma+\delta t_2}{\sqrt{2|\delta|}}$. Since the numerators represent the extinction coefficients $\tau(t_1)$ and $\tau(t_2)$ at the two endpoints of the ray segment, f_1 and f_2 are nonnegative real numbers. (If numerical inaccuracy results in a slightly negative value, it is replaced by zero.) Also, since δ is the slope of $\tau(t)$, $\delta > 0$ when $f_2 > f_1$.

Note that the *erfi* terms in (11) are multiplied by a factor of $e^{-f_2^2}$ if $\delta > 0$. When $erfi(f_2)$ is replaced by the Dawson integral from (12), the above factor $e^{-f_2^2}$ cancels the factor $e^{f_2^2}$ in $erfi(f_2) = \frac{2}{\sqrt{\pi}} e^{f_2^2} D(f_2)$. Thus the product $e^{-f_2^2}(erfi(f_2) - erfi(f_1))$ can be evaluated as $\frac{2}{\sqrt{\pi}}(D(f_2) - e^{f_1^2 - f_2^2} D(f_1))$. If $\delta > 0$, $f_2 > f_1$, so $f_1^2 - f_2^2$ is negative, and if it becomes so

negative that it leaves the valid domain of the exponential (causes underflow), then it is safe to replace $e^{f_1^2 - f_2^2}$ by zero.

If $\delta < 0$, then the corresponding product is $\frac{1}{i} e^{f_2^2} (erfi(\frac{f_2}{i}) - erfi(\frac{f_1}{i})) = e^{f_2^2} (erf(f_1) - erf(f_2))$. (The $\frac{1}{i}$ in front comes from the $\delta^{-2.5}$ factor in (11).) We use an approximation to $erf(x)$ for $x > 0$, given in [39] under the guise of the *complementary error function* $1 - erf(x)$, of the form $erf(x) \cong 1 - ue^{(-x^2 + p(u))}$ where $u = \frac{1}{1+0.5x}$, and $p(u)$ is a ninth degree Chebyshev polynomial selected to give an accurate fit to the tail of the error function. Thus we get $e^{f_2^2}(erf(f_2) - erf(f_1)) \cong u_2 e^{p(u_2)} - u_1 e^{f_2^2 - f_1^2} e^{p(u_1)}$, which avoids loss of accuracy when f_1 and f_2 are large, since the 1's cancel. In this case, $f_1 > f_2$, and we can again set $e^{f_2^2 - f_1^2}$ to zero if $f_2^2 - f_1^2$ is too negative.

Appendix B

Since the HIAC volume rendering system is currently used in batch mode, the input to the system is an image specification file which supplies all the details needed to generate the image. It has the advantage over a graphical user interface of providing an archive of the specifications of how the image was generated. In the following example image specification file, the # symbol indicates a comment.

```
RGBFILE    spxq.rgb
# file where rgba volume-rendered image to be written

PIXEL-SIZE 12
# write rgb file with 12 bits per pixel

DATAFILE    spx.vmag4

CMAPFILE    spxc1
# name of color map file

DMAPFILE    spxdvm4
# name of opacity (optical density) map file

WINDOW -5.517 5.517 -4.0 4.0
# bounds of viewplane window in data set coordinates

RESO 640 464
# viewplane window in pixels

CENTER
# automatically translate data so world coordinate
# origin at center of data set.

RX 21.3
RY 3.7
# rotate degrees about x and y axes.
```

```
PERSPECTIVE 100.0
# Use perspective projection, with eyeDistance = 100.

QUADRATIC
# use Gaussian integration for quadratic tetrahedra.

PROJTYPE 2
# Approximation method to use, if any, corresponds to methods M0-M3
# described in Section 7.

NEONSMOG
# use neon and smog glow energy treatment, default is williams and max

SURF 0.35  1. 0. 0.    .5   30.  1.
#    sfv   r  g  b     ks kexp trans  for Phong shaded semitransparent
#                                     colored isosurface

#      sfv = scalar field value for contour
#      r,g,b = color of surface
#      ks = fraction of light appearing at center of highlight
#      kexp = exponent determining size of highlight
#      trans = transparency of surface when viewed along normal

AMBIENT 0.2 0.2 0.2
# ambient light for surfaces

NORMAL 1.0
# specifies which side of surface is the outside of the contour

LIGHT 1 -0.5, 0.5, -1    0.8, 0.8, 0.8
# light number, vector (x,y,z) pointing towards light source located
# at infinity whose color is (r,g,b)

ZCLIP 0.0 3.0
# show z-clipped slab of the data parallel to the viewplane, with
#     znear= 0.0,    zfar=3.0

SUBPIXEL 2
# if projected cell covers less than 2 pixels, splat it over 3x3 square
# of nearest pixels using a quadratic B-spline.

RENDER
# commands for this image end here, now render it.
# other images may then be specified.

END
# end of specification file, no more images to render.
```

The input file for Figure 21 is:

```
datafile spx.vmag4
cmapfile spxc1
dmapfile spxdvm4
```

```
rgbfile fig1.rgb
window -6 6 -4.5 4.5
reso 560 420
quadratic
center
ry 95
rx 20
render
end
```

The color and optical density maps are:

```
file: spxc1
  COLORMAP
  -100 0 1 0
  0.15 0 1 0
  0.225 1 0 1
  0.3 1 0 0
  0.6 1 1 0
  1.5 1 1 1
  500 1 1 1
file: spxdvm4
  DENSITYMAP
  -500 .2
  .15 .2
  .2 .4
  .45 .4
  1 .6
  1.5 1
  500 1
```

The input file for Figure 22 is the same as above, except the line *quadratic* is replaced with *linear*. For Figure 23, the line is replaced with *approx*, and in Figure 24 the line *noslice* is added to that. The input file for Figure 29 is given below, showing the specification of the isosurface value and the lighting parameters.

```
rgbfile fig6.rgb
datafile spx.vmag4
cmapfile spxc1
dmapfile spxdvm4
window -5.517 5.517 -4.0 4.0
reso 640 464
quadratic
center
ry 20.0
surf 0.35  1. 0. 0.   .5 30. 1.
normal 1.
light 1 -0.5 0.5 -1 0.8 0.8 0.8
ambient 0.2 0.2 0.2
render
end
```

The input file for Figure 32 is similar to those above except for the addition of the subpixel splatting command *subpixel 2*, which means if a projected cell covers less than 2 pixels, the data is splatted over the 3x3 square of nearest pixels using a quadratic b-spline, rather than scan converting it.

Appendix C

This appendix describes the HIAC data structures for use with tetrahedral data sets. For quadratic tetrahedra, we assume the extra nodes are located on the edges. The data structures for zoo mesh data sets are described in Section 5.4.1.

Each scalar field in the data set is stored in a floating point array with one element per node, in the same ordering as is used for the *xyz* array, described below. In addition to the data arrays, three other basic arrays are used: *elems*, *nodes*, and *xyz*. The elements of the *elems* and the *nodes* arrays are integer values and the elements of the *xyz* array are three-tuples of floating point values. Each cell has one entry in the *elems* array, its total number of nodes *nn*. However rather than encoding *nn* directly, the *elems* array stores the cumulative total of *nn*. So cell i will have $nn = elems[i] - elems[i-1]$ nodes. The nodes for cell i are in the *nn* entries in the *nodes* array starting with $nodes[elems[i]]$. The nodes stored in the *nodes* array are pointers to the coordinates of the nodes which are stored in the *xyz* array. The conventional nodes are specified first in the *nodes* array in a standard order, followed by the interior nodes if any. We also keep a flag indicating whether the data set is linear, quadratic or cubic. This flag disambiguates different types of cells with the same number of nodes, e.g. a quadratic brick has the same number of nodes as a cubic tetrahedron. We assume elements of different orders (linear, quadratic, cubic) will not be combined in one mesh.

References

[1] J. Bennett, R. Cook, N. Max, D. May, and P. Williams, "Parallelizing a High Accuracy Hardware-Assisted Volume Renderer for Meshes with Arbitrary Polyhedra," submitted to *Proceedings IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, Oct. 2001.

[2] P. Cignoni, C. Montani, D. Sarti, and R. Scopigno, "On the Optimization of Projective Volume Rendering," *Proc. Eurogr. Wkshop., Vis. in Sci Comput. '95* R. Scaneni, J. van Wijk, and P Zanarini, eds., pp 59–71, May 1995

[3] H. E. Cline, W. E. Lorensen, S. Ludke, C. R. Crawford and B. C. Teeter, "Two Algorithms for the three-dimensional reconstruction of tomograms," *Medical Physics* vol. 15, no. 3, pp. 64–72, May 1988.

[4] J. Comba, J. Klosowski, N. Max, J. Mitchell, C. Silva, P. Williams, "Fast Polyhedral Cell Sorting for Interactive Rendering of Unstructured Grids," *Eurographics '99*.

[5] R. Cook, N. Max, C. Silva and P. Williams, "Efficient and Exact Visibility Sorting of Zoo-Mesh Data Sets,", submitted to *IEEE Visualization 2001*, Oct. 2001.

[6] M. de Berg, M. Overmars, and O. Schwarzkopf, "Computing and Verifying Depth Orders," *SIAM J. Comput.* vol. 23, pp. 437–446, April 1994.

[7] R. Farias, J. Mitchell and C. Silva, "ZSWEEP: An Efficient and Exact Projection Algorithm for Unstructured Volume Rendering," *Vol. Vis. Sym. 2000*, pp 91–99, Oct. 2000.

[8] R. Farias and C. Silva, "Out-of-Core Rendering of Large Unstructured Grids," to appear in *IEEE Comp. Graph. and App.*, 2001.

[9] R. Farias and C. Silva, "Parallelizing the ZSWEEP Algorithm for Distributed-Shared Memory Architectures," to appear in *Intl. Workshop on Vol. Graph.*, 2001.

[10] J. Foley, A. van Dam, S. Feiner, and J. Hughes, "Computer Graphics Principles and Practice," 2nd Edition, Addison-Wesley, 1990.

[11] T. Frühauf, "Raycasting of Nonregularly Structured Volume Data,", Computer Graphics Forum, vol 13, no. 3, pp 295–303, 1994.

[12] T. Frühauf, "Raycasting with Opaque Isosurfaces in Nonregularly Structured CFD Data,", in *Visualization in Scientific Computing '95*, R. Scateni, J. van Wijk and P. Zanarini, eds., Springer Verlag, 1995.

[13] R. Gallagher and J. Nagtegaal, "An Efficient 3D Visualization Technique for Finite Element Models and Other Coarse Volumes," *Computer Graphics*, vol. 23, no. 3, pp. 185–192, July 1989.

[14] M. P. Garrity, "Raytracing Irregular Volume Data," *Computer Graphics*, vol. 24, no. 5, pp. 35–40, Nov. 1990.

[15] C. Giertsen, "Volume Visualization of Sparse Irregular Meshes," *Computer Graphics*, vol. 12, no. 2, pp. 40–48, Mar. 1992.

[16] C. Giertsen and A. Tuchman, "Fast Volume Rendering with Embedded Geometric Primitives," *Visual Computing - Integrating Computer Graphics with Computer Vision*, T.L.Kunii (ed), Springer Verlag, pp. 253–271, 1992.

[17] R. Haimes, "Visual3: Interactive Unsteady Unstructured 3D Visualization," *AIAA Paper 91-0794*, Reno NV, Jan. 1991.

[18] M. S. Karasick, D. Lieber, L. R. Nackman, and V. T. Rajan, "Visualization of Three-Dimensional Delaunay Meshes," *Algorithmica* vol. 19, pp. 114–128, 1997.

[19] H. Kardestuncer and D. H.Norrie, "Finite Element Handbook," McGraw-Hill, New York, 1987.

[20] K. Koyamada, "Volume Visualization for the Unstructured Data," *SPIE Vol. 1259 Extracting Meaning from Complex Data: Processing, Display, Interaction*, 1990.

[21] W. E. Lorensen and H. E. Cline, "Marching Cubes: A High Resolution 3-D Surface Construction Algorithm," *Computer Graphics*, vol. 21, no. 4, pp. 163–169, July 1987.

[22] B. A. Lucas, "A Scientific Visualization Renderer," *Proc. Visualization '92*, Boston, pp. 227–234, Oct. 1992.

[23] K-L. Ma, "Parallel Volume Ray-Casting for Unstructured-Grid Data on Distributed-Memory Architectures," *ACM Parallel Rendering Symposium*, pp. 23–30, Oct 1995.

[24] R. H. MacNeal, "Finite Elements: Their Design and Performance," Marcel Dekker, Inc, NY, 1994.

[25] X. Mao, "Splatting of Nonrectilinear Volumes Through Stochastic Resampling," *IEEE Trans. on Visualization and Computer Graphics*, vol 2, no. 2, pp. 156–170, June 1996.

[26] N. Max, P. Hanrahan and R. Crawfis, "Area and Volume Coherence for Efficient Visualization of 3D Scalar Functions," *Computer Graphics*, vol. 24, no. 5, pp. 27–33, Nov. 1990.

[27] N. Max, "Optical Models for Direct Volume Rendering," *IEEE Trans. on Visualization and Computer Graphics*, vol. 1, no. 2, pp. 99–108, June 1995.

[28] N. Max, P. Williams, and C. Silva, "Approximate Volume Rendering for Curvilinear and Unstructured Grids by Hardware-Assisted Polyhedron Projection," *International Journal of Imaging Systems and Technology*, Vol 11, pp. 53–61, 2000.

[29] N. Max, P. Williams, C. Silva, "Cell Projection of Meshes with Non-Planar Faces," submitted to *Proceedings Scientific Visualization Workshop Dagstuhl 2000*, April, 2000.

[30] S. Molnar, M. Cox, D. Ellsworth and H. Fuchs, "A Sorting Classification of Parallel Rendering," *IEEE Computer Graphics and Applications*, pp. 23–32, July, 1994.

[31] S. Molnar, J. Eyles, and J. Poulton, "PixelFlow: High-Speed Rendering Using Image Composition," *Computer Graphics*, 26(2) pp. 231–240, July 1992.

[32] K. Mueller and R. Yagel, "Fast Perspective Volume Rendering with Splatting Utilizing a Ray-Driven Approach," *Proc. Visualization '96*, pp. 65–72, Oct 1996.

[33] M. Newell, R. Newell and T. Sancha, "Solution to the Hidden Surface Problem," Proc ACM National Conference, 1972, pp. 443–450.

[34] M. Newell, "The Utilization of Procedure Models in Digital Image Synthesis," Ph.D. Thesis, University of Utah, 1974 (UTEC-CSc-76-218 and NTIS AD/A 039 008/LL).

[35] G. Nielson and J. Sung, "Interval Volume Tetrahedralization," *Proc. IEEE Vis. 1997*, pp. 221–228, 1997.

[36] K. Novins and J. Arvo, "Controlled Precision Volume Integration," *Proc. 1992 Workshop Volume Visualization*, Boston, pp. 83–89, Oct. 1992.

[37] J. O'Rourke, "Computational Geometry in C", Cambridge University Press, 1995.

[38] C. E. Prakash, "Parallel Voxelization Algorithms for Volume Rendering of Unstructured Grids," PhD Thesis, Supercomputer Centre, Indian Institute of Science, 1996.

[39] W. Press, S. Teukolsky, W. Vetterling and B. Flannery "Numerical Recipes in Fortran," Cambridge University Press, 1992.

[40] D. F. Rogers, "Procedural Elements for Computer Graphics," McGraw-Hill, New York, 1985.

[41] G. B. Rybicki, "Dawson Integral and the Sampling Theorem," *Computers in Physics*, vol. 3, no. 2, pp. 85–87, 1989.

[42] P. Sabella, "A Rendering Algorithm for Visualizing 3D Scalar Fields," *Computer Graphics*, vol. 22, no. 4, pp. 51–58, Aug. 1988.

[43] R. Sedgewick, "Algorithms in C++," Addison-Wesley 1992, pp. 359–371

[44] M. Segal and K. Akeley, "The OpenGL Graphics System: A Specification," Silicon Graphics, Inc., April 1999.

[45] G. Schussman and N. Max, "Hierarchical Perspective Volume Rendering using Triangle Fans," to appear ino *Volume Graphics 2001*, June 2001.

[46] P. Shirley and A. Tuchman, "A Polygonal Approximation to Direct Scalar Volume Rendering," *Computer Graphics*, vol. 24, no. 5, pp. 63–70, Nov. 1990.

[47] "Silo User's Guide, Revision 1," Lawrence Livermore National Laboratories Tech. Report UCRL-MA-118751, August 2000.

[48] C. Silva and J. S. B. Mitchell, "The Lazy Sweep Ray Casting Algorithm for Rendering Irregular Grids," *IEEE Trans. on Visualization and Computer Graphics*, vol. 3, no. 2, pp. 142–157, June 1997.

[49] C. Silva, J. Mitchell and P. Williams, "An Exact Interactive Time Visibility Ordering Algorithm for Polyhedral Cell Complexes," *Proceedings ACM/IEEE Volume Visualization Symposium*, Oct. 1998.

[50] C. Stein, B. Becker and N. Max, "Sorting and Hardware Assisted Rendering for Volume Visualization," *Proc. 1994 Symposium on Volume Visualization*, Washington, DC, pp. 83–90, Oct. 1994.

[51] K. Subramanian and D. Fussell, "Applying Space Subdivision Techniques to Volume Rendering," *IEEE*, pp 150–158, 1990.

[52] J. Swan, K. Mueller, T. Moller, N. Shareef, R. Crawfis and R. Yagel, "An Antialiasing Technique for Splatting," *Proc. IEEE Visualization '97*, Phoenix, Az., pp. 197–204, Oct. 1997.

[53] S. Krishnan, C. Silva, and B. Wei, "A Hardware-Assisted Visibility Ordering Algorithm with Applications to Volume Rendering," to appear in *VisSym '01*, 2001.

[54] L. Westover, "Interactive Volume Rendering," *Proc. 1989 Workshop Volume Visualization*, Chapel Hill, pp. 9–16, May 1989.

[55] J. Wilhelms and A. Van Gelder, "A Coherent Projection Approach for Direct Volume Rendering," *Computer Graphics*, vol. 25, no. 4, pp. 275–284, July 1991.

[56] J. Wilhelms, "Pursuing Interactive Visualization of Irregular Grids," *The Visual Computer*, vol. 9, pp. 450–458, 1993.

[57] J. Wilhelms, A. Van Gelder, P. Tarantino and J. Gibbs, "Hierarchical and Parallelizable Direct Volume Rendering for Irregular and Multiple Grids," *Proc. Visualization '96*, pp. 57–64, Oct. 1996.

[58] P. L. Williams, "Visibility Ordering Meshed Polyhedra," *ACM Trans. on Graphics*, vol. 11, no. 2, pp. 103–126, April 1992.

[59] P. L. Williams, "Interactive Splatting of Nonrectilinear Volumes," *Proc. Visualization '92*, Boston, pp. 37–44, Oct. 1992.

[60] P. L. Williams and N. L. Max, "A Volume Density Optical Model," *Proc. 1992 Workshop Volume Visualization*, Boston, pp. 61–68, Oct. 1992.

[61] P. Williams and S. Uselton, "Metrics and Generation Specifications for Comparing Volume Rendered Images," *Journal Visualization and Computer Animation*, Vol 10, pp. 159–178, 1999.

[62] P. Williams, N. Max, and C. Stein, "A High Accuracy Volume Renderer for Unstructured Data," *IEEE Trans. on Vis. and Comp. Gr. 4(1)*, pp. 37–54, 1998.

[63] P. H. Winston and B. K. P. Horn, "Lisp, second edition," Addison Wesley, Reading MA, (1984).

[64] C. M. Wittenbrink, "Irregular Grid Volume Rendering with Composition Networks," *proc. IS&T/SPIE Visual Data Exploration and Analysis V*, San Jose, CA, Jan. 1998.

[65] C. M. Wittenbrink, "CellFast: Interactive Unstructured Volume Rendering," *proc. IS&T/SPIE Visual Data Exploration and Analysis V*, San Jose, CA, Jan. 1998.

[66] M. Woo, J. Neider and T. Davis, "OpenGL Programming Guide, second edition," Addison Wesley, p. 372, 1997.

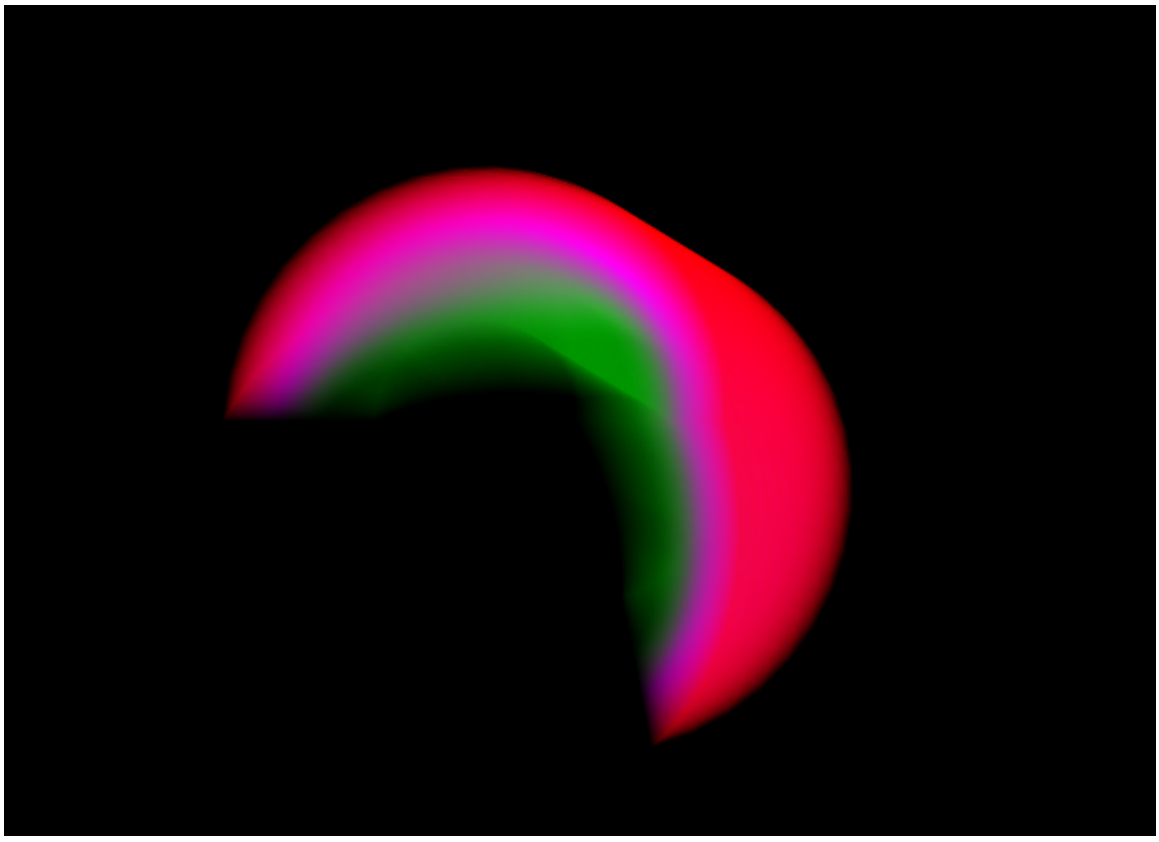

Figure 20: Volume rendering of a twisted curvilinear zoo mesh using view-dependent subdivision

[67] R. Yagel, D. Reed, A. Law, P-W. Shih and N. Shareef, "Hardware Assisted Volume Rendering of Unstructured Grids by Incremental Slicing," *Proc. 1996 Symposium on Volume Visualization*, pp. 55–62, Nov. 1996.

[68] "Handbook of Chemistry and Physics," Chemical Rubber Publishing Co., Cleveland, Ohio.

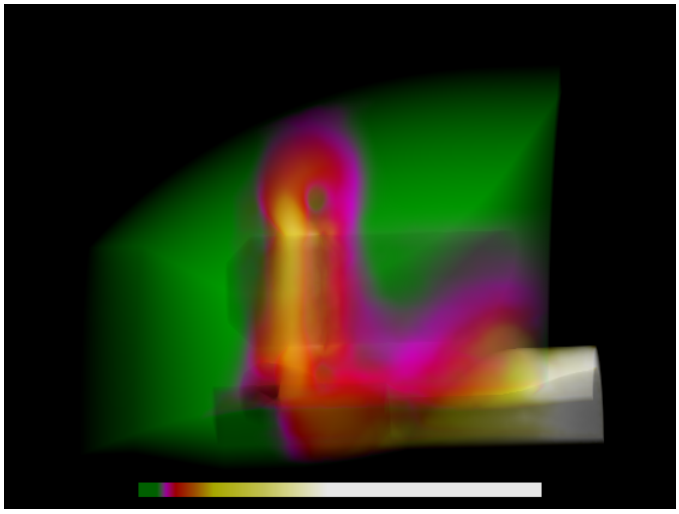

Figure 21: Volume rendering of coolant velocity magnitude in a nuclear reactor using the exact integration method for quadratic tetrahedra.

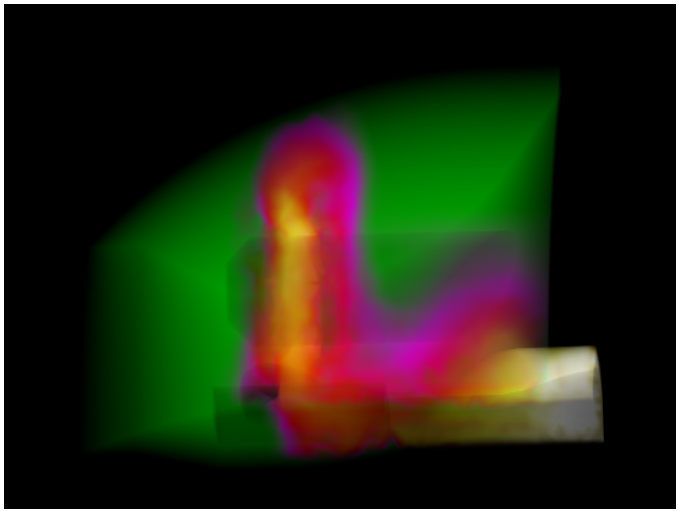

Figure 22: Same data set as previous image but rendered using the exact integration method for linear tetrahedra, i.e. the data at the interior nodes was neglected.

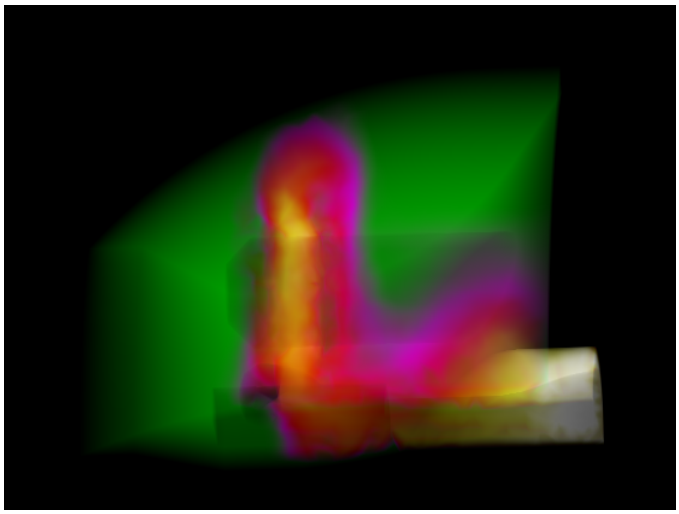

Figure 23: Same data set as previous image, but rendered using the approximate integration method described in Section 4.

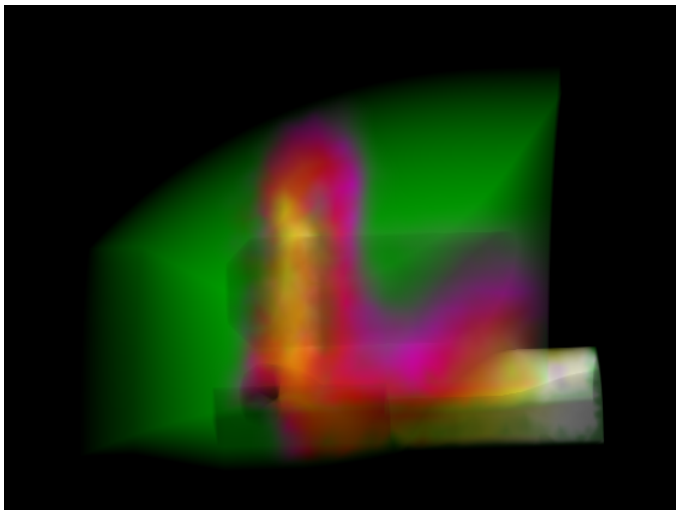

Figure 24: Same data set as previous image, but rendered using the approximate integration method, without slicing the cells into slabs.

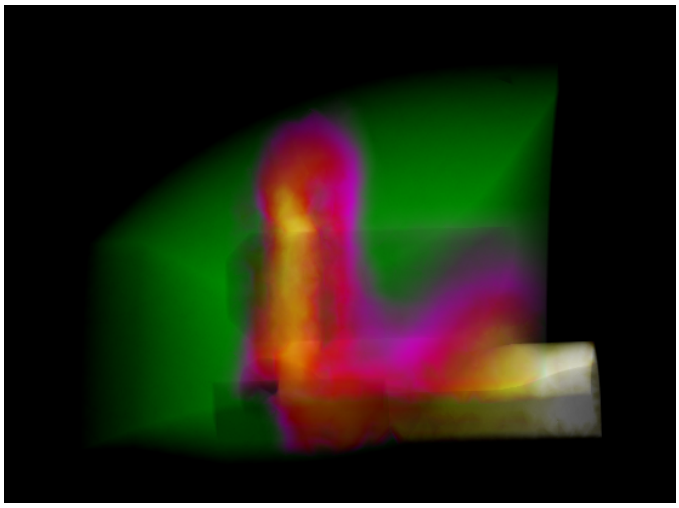

Figure 25: Same data set as previous image, but rendered using the hardware-based polyhedron projection method (M3) described in Section 6.

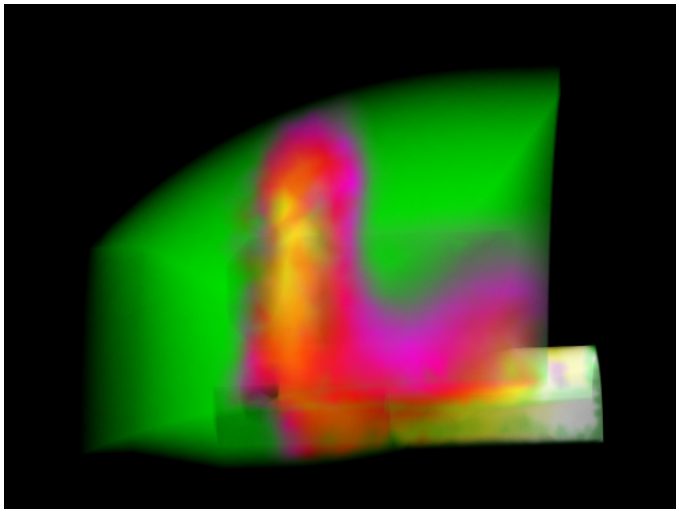

Figure 26: Same data set as previous image, but rendered using the hardware-based polyhedron projection method (M2) described in Section 6.

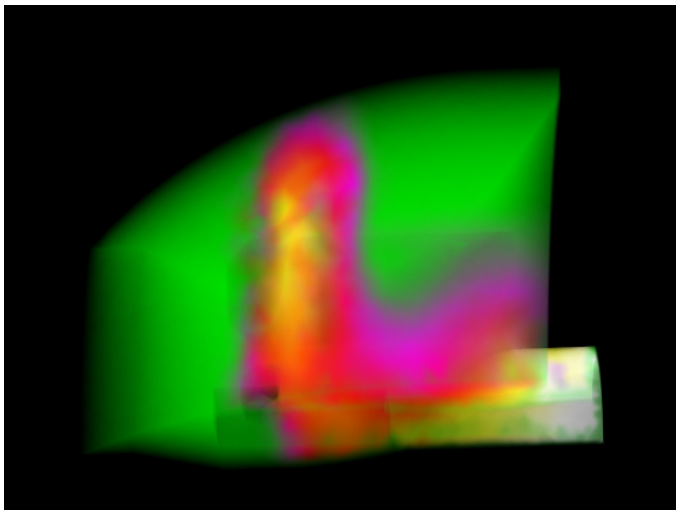

Figure 27: Same data set as previous image, but rendered using the hardware-based polyhedron projection method (M1) described in Section 6.

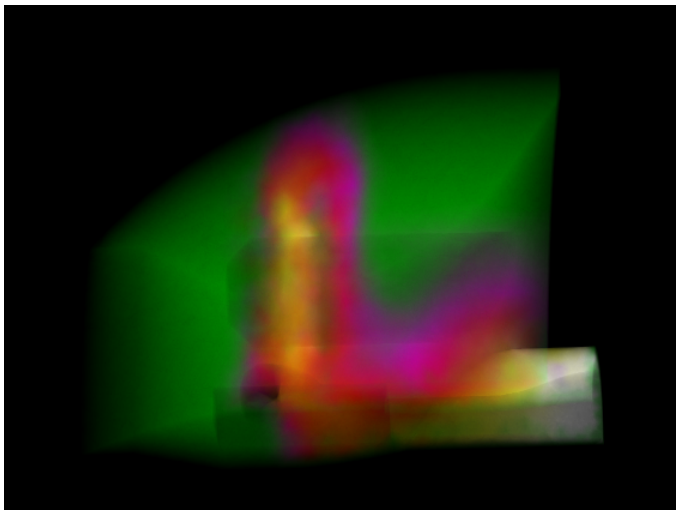

Figure 28: Same data set as previous image, but rendered using the hardware-based polyhedron projection method (M0) described in Section 6.

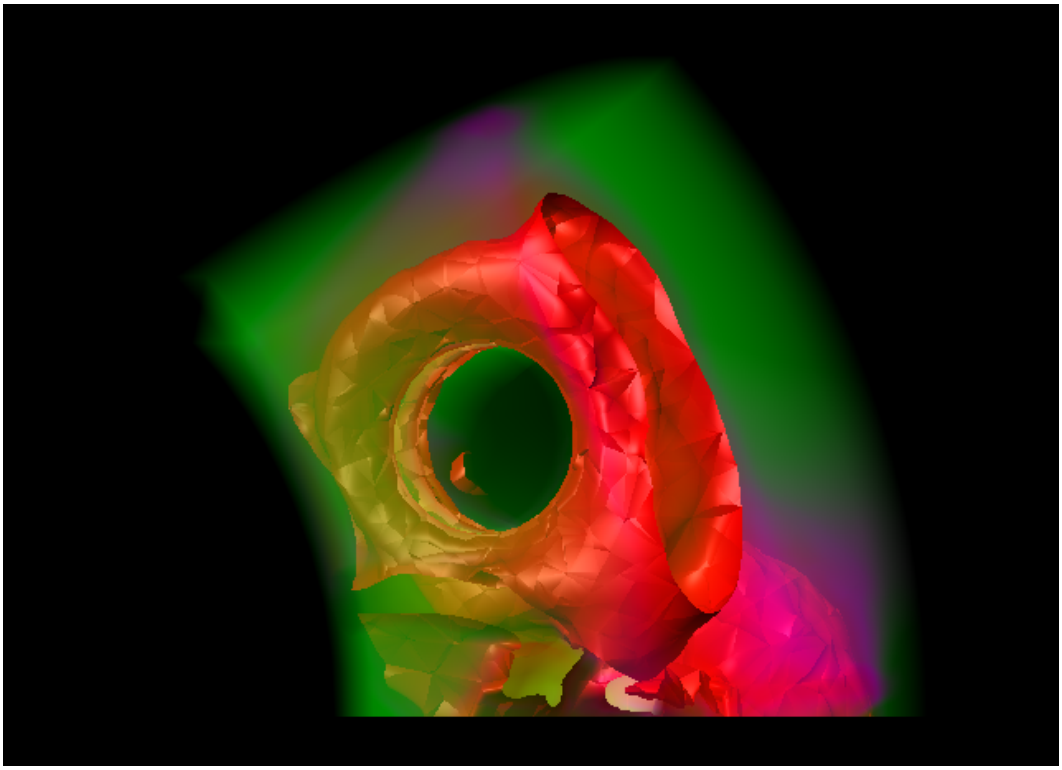

Figure 29: Volume rendering of coolant velocity magnitude in a nuclear reactor using the exact integration method for quadratic tetrahedra with embedded semitransparent illuminated isosurfaces.

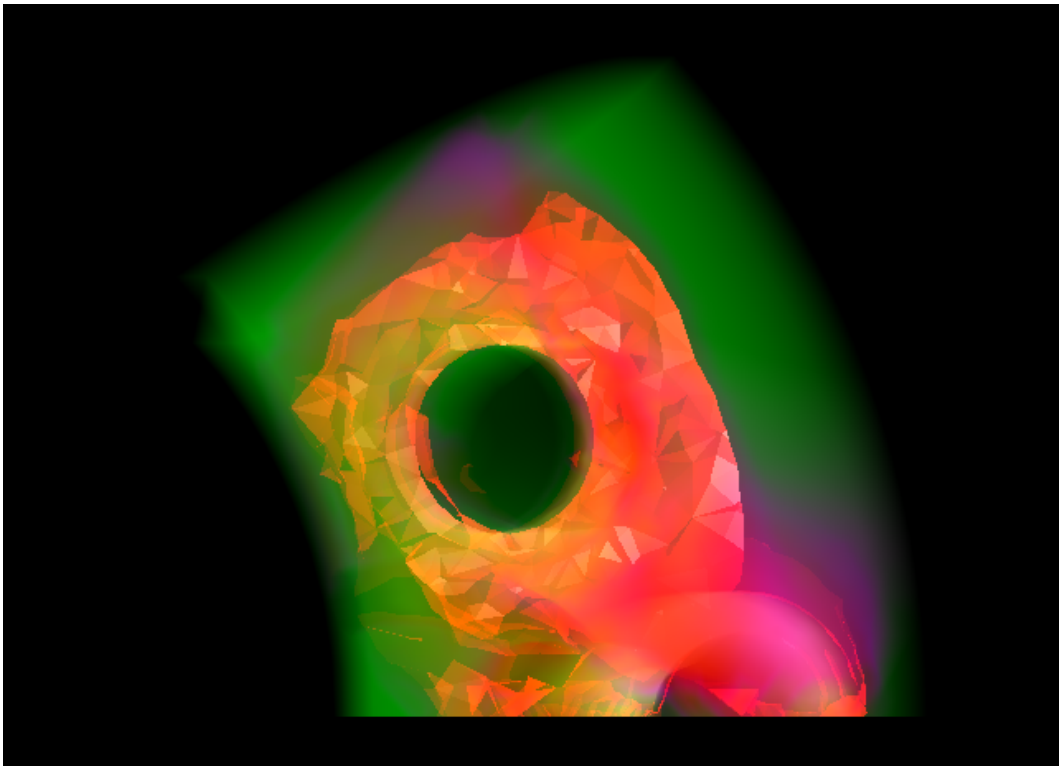

Figure 30: Volume rendered image of same data set and isosurfaces as previous figure, but using the integration method for linear tetrahedra.

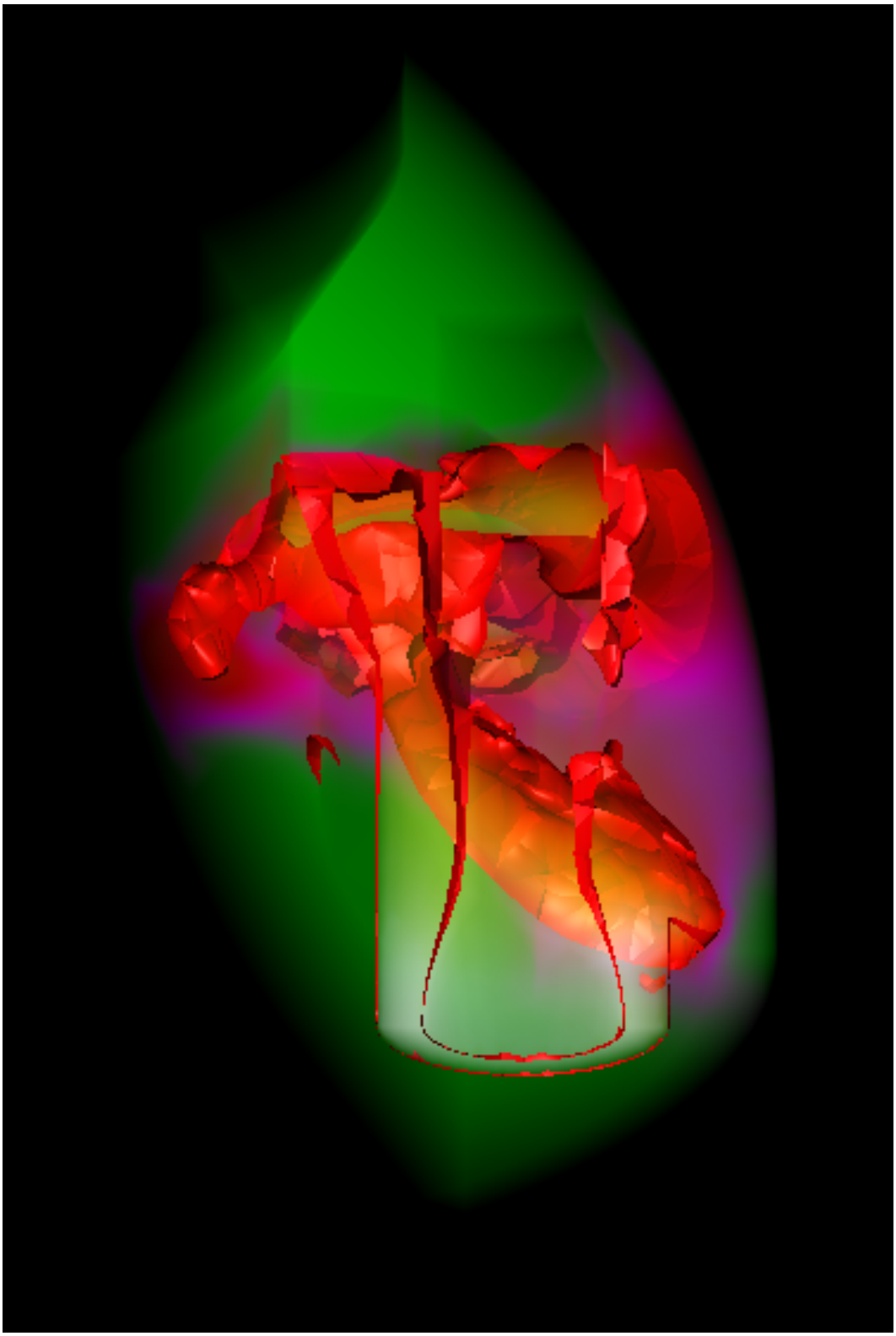

Figure 31: Same data set and isosurfaces as previous image, but from a different viewpoint. The exact integration method for quadratic tetrahedra is used.

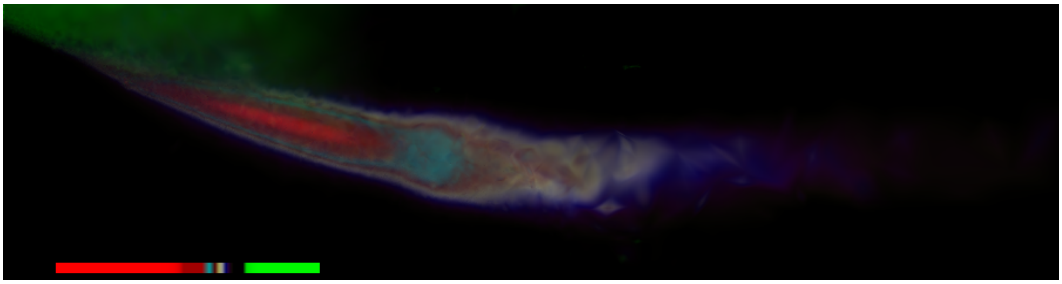

Figure 32: Volume rendered image of air flow past an *F-177a* aircraft wing using the exact method method for linear tetrahedra, with subpixel splatting

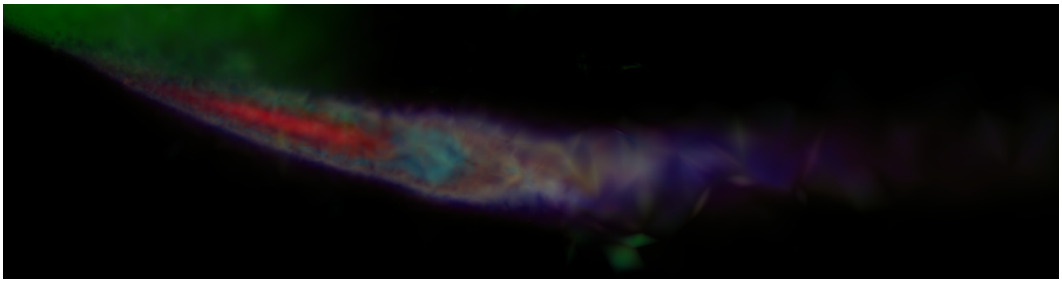

Figure 33: Same data set and viewing parameters as previous image, but using the approximate method and no subpixel splatting.

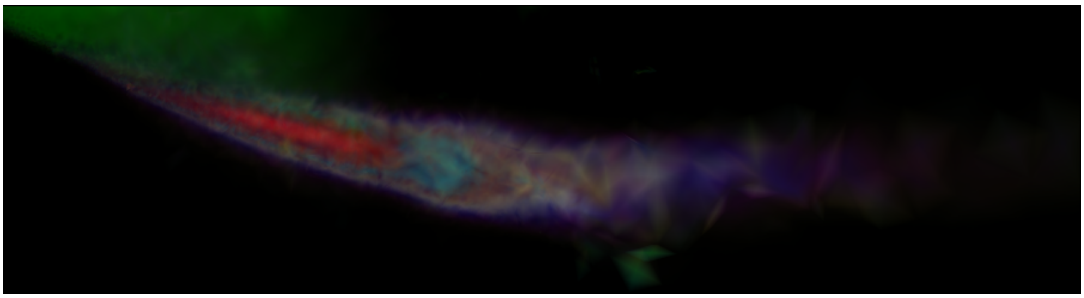

Figure 34: Same data set and viewing parameters as previous image, but rendered in parallel with the tetrahedra-only mode, using the M0 hardware-assisted projection algorithm.

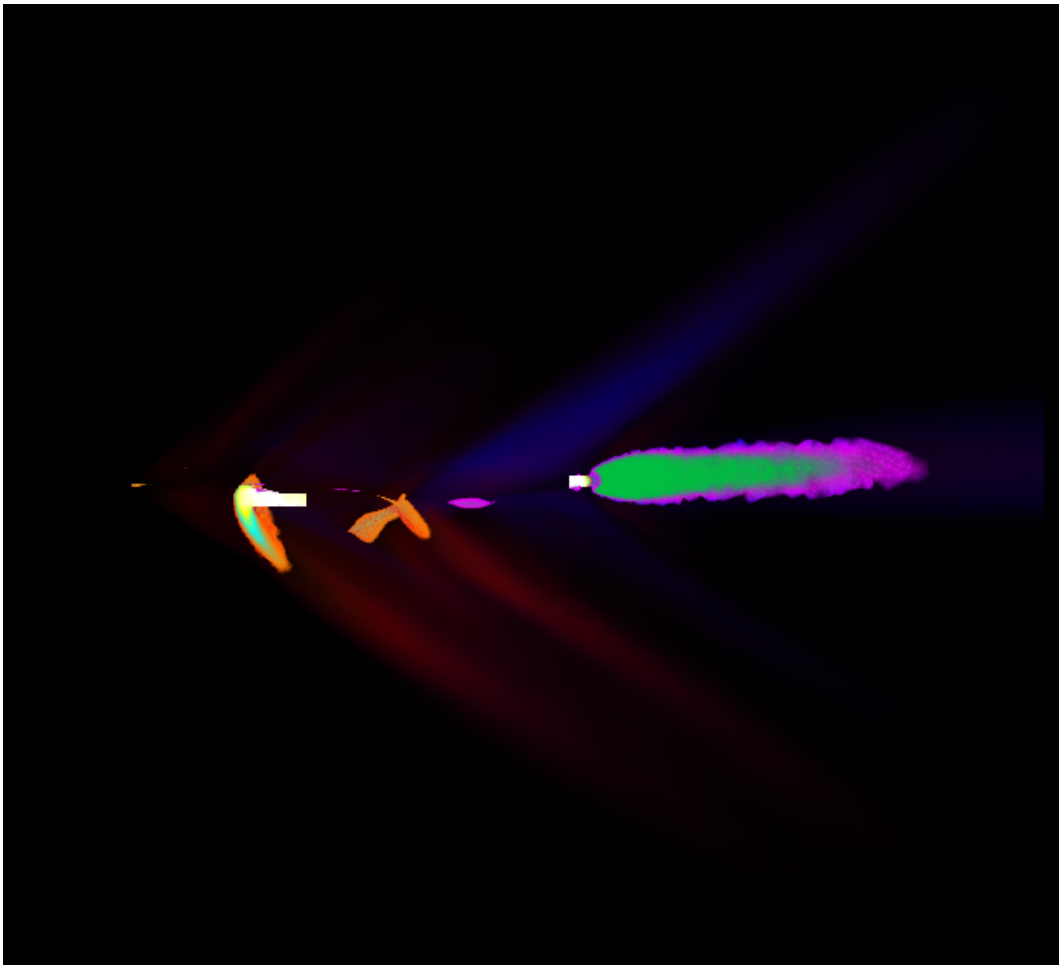

Figure 35: The *fighter* data set rendered in parallel in tetrahedra-only mode, using the M0 hardware-assisted projection algorithm.

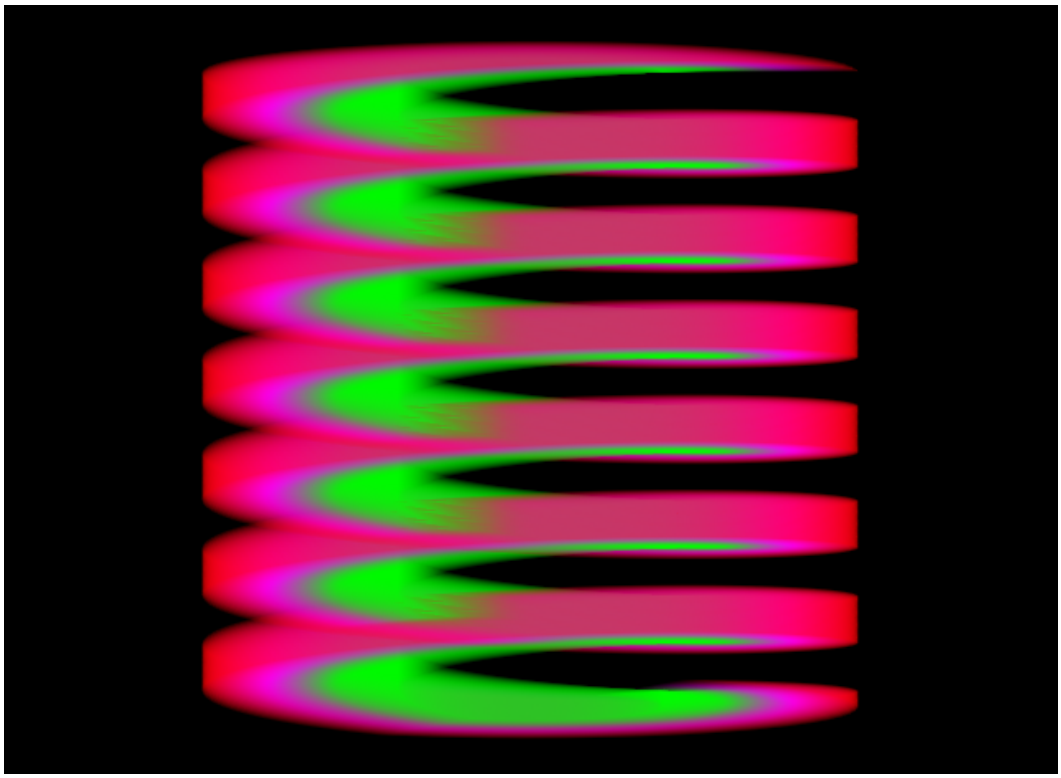

Figure 36: The *helix* silo data set rendered in parallel in tetrahedra-only mode using the M0 hardware assisted projection algorithm, after first decomposing all elements into tetrahedra.

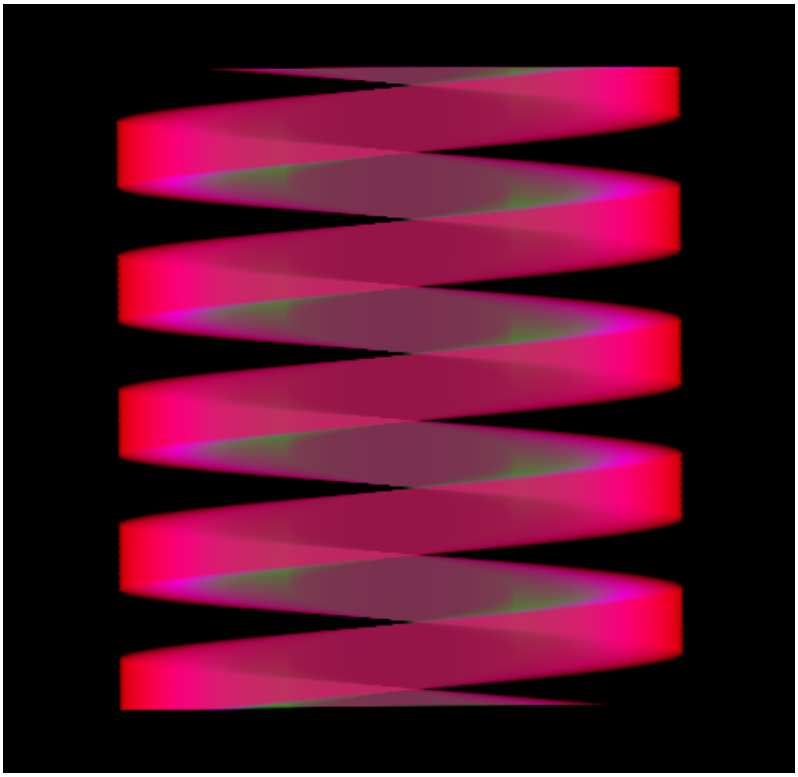

Figure 37: Different view of same data set as previous image but created in zoo-mode in software, the most accurate method for zoo mesh data.